UNIVERSITY OF OSLO
Department of Informatics

Increasing the efficiency of a file server by removing redundant data transfers in popular downloads

Master thesis

Dag Henning
Liodden Sørbø

August 2013

Increasing the efficiency of a file server by removing redundant data transfers in popular downloads

Dag Henning Liodden Sørbø

August 2013

Abstract

The standard method of transferring files from a file server to clients in the Internet is through TCP connections. The whole file is transferred separately to each client via a unicast connection. It often happens that clients are downloading the same file concurrently within a certain time interval. During this interval, the server transfers multiple copies of the same data. This creates redundant data transfers in the network. In this thesis we present the CacheCast file server, which removes these redundant data transfers by utilizing the newly developed CacheCast mechanism. This mechanism removes redundancy from single source multiple destination transfers.

In order to benefit from the CacheCast mechanism, the same data chunk must be transferred to multiple clients within a short time frame. CacheCast caches payloads on routers in the network, such that equal payloads are transferred only once over each link. In a live streaming system, all clients consuming the same video or voice stream are receiving the same data synchronously. Thus, live streaming systems can greatly benefit from CacheCast. In a file server, the clients are not synchronized *per se*. CacheCast support in a file server therefore requires a special system design. The key idea in the CacheCast file server is to reorder the file blocks before transmission, such that the same file block is transferred to multiple clients. CacheCast is then able to remove the redundant data transfers.

This thesis includes the design, implementation and evaluation of the CacheCast file server. The system is implemented in the ns-3 network simulator, in order to perform experiments in a network with dozens of clients. Three major aspects of the system are evaluated, namely the effects on the bandwidth consumption in the network, the impact on the download time experienced by the clients, and the fairness among concurrently connected clients. The performance of the CacheCast file server is compared against the performance of an FTP server.

The evaluation has revealed that the CacheCast file server performs significantly better than an FTP server, which transfers the files using TCP. It delivers the files faster to the receivers, and reduces the total bandwidth consumption in the network. In our experiments, the download time is reduced by a factor of 10 and the bandwidth consumed is 89 % less then when using an FTP server. These performance gains are attributed to the CacheCast support in the file server. The evaluation also shows that the CacheCast file server ensures fairness among competing clients.

Acknowledgments

First and foremost I would like to thank my two marvelous supervisors Thomas Plagemann and Piotr Srebrny. Piotr, thank you for all the fruitful technical discussions we had and for always pointing me in the right direction. Thomas, thank you for your wise and clear suggestions, especially throughout the writing process. Without your help and guidance it would not have been possible to complete this thesis. I am sincerely grateful for the work you have done.

To my dear wife Liv Solveig - without your support, love, patience, forgiveness, tolerance, indulgence, understanding and kindness, I could not have finished this thesis. I cannot express strongly enough how thankful I am.

To my sweet newborn daughter Oline - thank you for staying inside the womb until due date, so I could finish most of my thesis by then. And thank you for being such a good sleeper.

I would also like to thank the rest of my family and my friends for the great support and motivating words in time of need.

Last but not least, I thank my heavenly Father for his never-ending grace, and for keeping me as his child through it all.

Soli Deo gloria

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Over the last two decades the Internet has become immensely popular, especially after the introduction of the World Wide Web (WWW). Various content is distributed and consumed all over the globe with an ever increasing pace. As the number of users connected to the Internet grows, more and more people access the same content, often within a short time interval. We give three examples: (1) Many people are browsing the same news site during the lunch break. (2) Whenever a user with many followers on YouTube uploads a new video, many of his followers view the video shortly after it has been uploaded. (3) If a new version of a popular software is released, many users are upgrading to the new version within a short time frame.

A majority of the content available on the Internet is accessed using various file transmission protocols. In order for the reader of a news site to view the site's contents, multiple files are transferred to his computing device. Similarly, when a user is upgrading a certain software, a new installation file is downloaded to his computer. The standard protocols for file transfer in the current Internet use the *Transmission Control Protocol (TCP)* [37] for the data transmission. TCP is built for single source single destination transfer. Thus, two hosts communicate using a single TCP connection. If a file is downloaded by multiple clients there exists a TCP connection for each client. The file is transferred once per connection.

When many users are requesting the same file within a short time period, there are overlaps between the downloads. This means, during an overlapping time period the same data file is transferred from the server to multiple clients. Since the whole file is transferred once over each unicast connection, this creates redundant data transfers in the network. In this thesis we explore the possibility of removing this redundancy.

A new mechanism called CacheCast [43] has been developed, which removes redundant payloads from packets traversing the same links in a network. In order to benefit from CacheCast, the same data chunk must be transferred to multiple clients. In a live streaming application, the connected clients are synchronized in time, such that the same live data stream can be transferred to all clients. Therefore, in live streaming systems, CacheCast is able to remove much redundancy and its performance is close to the performance of IP multicast [15].

However, when downloading files, the clients are not synchronized in time. A user can start a download at any point in time. When there are multiple clients downloading the same file concurrently, each client is receiving a different part of the file, since they have started the download at different points in time. Hence, to benefit from CacheCast, a new mechanism is necessary to enable transmission of the same data chunk to all clients. In this thesis we design, implement and evaluate such a mechanism.

1.1 Problem statement

As briefly mentioned, when multiple clients are downloading the same file at approximately the same time, there are time periods where the downloads overlap, i.e. the clients are downloading the file concurrently. During these overlapping time periods the same file is transferred to these clients. However, since the file is transferred once per client connection the same data traverse the network multiple times. This results in multiple redundant data transfers in the network. The goal of this thesis is to design a file server which removes much of this redundancy by using the CacheCast mechanism. Throughout the thesis this file server is called *CacheCast file server*.

1.2 Methods

In this thesis we take advantage of different methods and use different approaches to address the various parts of the thesis. Here, we briefly introduce the methods used in this thesis.

First, we study both the CacheCast mechanism and the specifics of file transmission and analyse the requirements for the CacheCast file server. This requirement analysis is performed to point out which parts of standard file transmission (using TCP) can be altered to support the CacheCast mechanism. Based on the requirement analysis we create a system design and a CacheCast file server architecture.

The CacheCast file server is implemented in the *ns-3* network simulator in order to test the behavior of the system in a simulated network. Multiple experiments are designed and run to measure how the CacheCast file server impacts the file transfer and the network resources. The results of the simulations are analysed and discussed in an evaluation part. We compare the performance of the CacheCast file server to the *de facto* standard file transfer protocol in the Internet, namely FTP [38].

1.3 Thesis contributions

There are two major contributions of this thesis; the design of a file server with CacheCast support and a performance evaluation of this file server.

CacheCast file server design

The CacheCast file server design is fundamentally different from the design of an FTP server. While an FTP server is designed for single source single destination transfer, the CacheCast file server is designed for single source multiple destination transfer. The transmission of a file is not done in a sequential manner, but the transmission order of the different parts of the file is modified to enable transmission of the same data chunk to multiple clients. CacheCast is then able to remove the redundant data transfers introduced by the transmission of the same data on each unicast connection.

Performance evaluation

The evaluation in this thesis compares the performance of the CacheCast file server to the performance of an FTP server. The three major outcomes from the evaluation are as follows: (1) The CacheCast file server is able to remove redundant data transfers from the network, which significantly decreases the bandwidth consumption in the network. (2) A consequence of the redundancy removal is reduced download time. When there are many overlapping clients the download time of a file is significantly reduced. (3) The CacheCast file server ensures fair share of the bandwidth capacity among competing clients.

1.4 Related work

There have been great research efforts in the area of efficient and reliable data transmission to multiple receivers. Much of this research is based on multicast techniques, such as *IP multicast* [15] and *Application Layer Multicast* [26]. Many different protocols for reliable single source multiple destination file transfer have been developed, such as *Reliable Multicast Protocol* [48]. However, IP multicast has not been widely deployed in the Internet, due to various reasons [16]. Therefore, other systems based on overlay networks and peer-to-peer communication, such as BitTorrent [39], have been developed.

In order to throughly compare the CacheCast file server to other related work, insights into the specifics of the design and functionality of the CacheCast file server are required. Therefore, we present a discussion of related work and a comparison to the CacheCast file server in Chapter 7.

1.5 Thesis structure

The structure of this thesis is as follows. In this chapter, a general introduction into the problem area is given. In Chapter 2, background information and necessary knowledge of existing network technologies are presented. Chapter 3 contains a discussion of basic design considerations for a file server with CacheCast support. These design considerations are transformed into a system architecture, which is explained in Chapter 4.

In Chapter 5, we introduce and discuss the implementation of the system. The evaluation of the CacheCast file server is presented in Chapter 6. Here we explain the performed experiments and present and analyse the results. In Chapter 7, we present some work related to the CacheCast file server. Chapter 8 summarizes the thesis.

Chapter 2

Background

The description of the CacheCast file server design requires some insight into existing network technologies. In this chapter we present the necessary background information to understand the discussion and reasoning throughout this thesis.

In the first section we carefully describe the CacheCast mechanism. It is the basis for the functionality of the CacheCast file server, so a thorough understanding of this mechanism is crucial. Next, we introduce the concept of a transport protocol, in order to prepare for the section on reliability. Reliable transfer is the main requirement for file transmission, so in Section 2.3 we describe different approaches for meeting this requirements. The CacheCast file server is highly dependent on the popularity of files. Therefore, in Section 2.4 we analyse how users on the Internet consume content. In the last section of this chapter we briefly introduce the ns-3 network simulator. In this thesis both the CacheCast mechanism and the CacheCast file server are implemented in ns-3.

2.1 CacheCast

CacheCast [44] is a new technique of removing redundant data transfers on a link[1] when multiple receivers are receiving the same content. The purpose of CacheCast is to remove as much as possible of the overhead when using many unicast connections for the same data over the same link. To benefit from CacheCast, the same data has to be transferred to multiple receivers in a batch. In short, CacheCast only sends the packet payload once over a link together with the packet headers, and the responsibility lies on the router on the link exit to forward the payload to each receiver. With this redundancy removal, CacheCast achieves close to multicast performance when the numbers of receivers grow large, without introducing new protocols into the Internet.

[1] In this context a link is the physical transport medium between two hosts in a network.

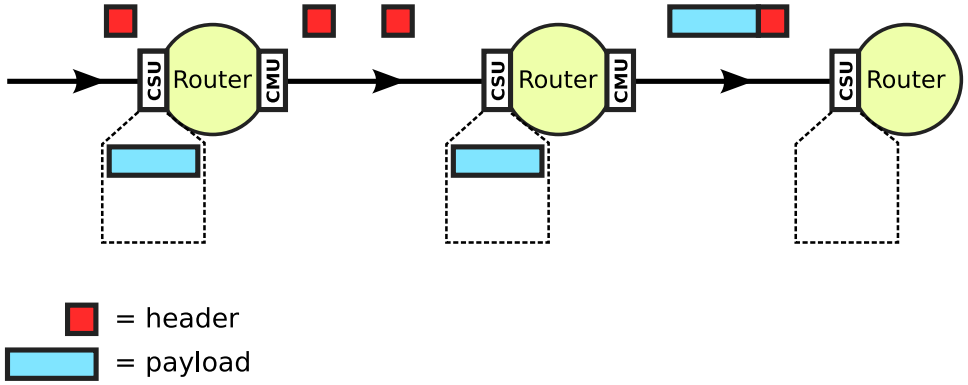

Figure 2.1: Overview of the CacheCast caching mechanism

2.1.1 CacheCast specifics

As the name implies, CacheCast uses caching to remove the redundant data packets from a link. When the same packet payload is transferred multiple times over the same link, CacheCast removes all payloads besides the first. This first packet payload is cached on the router on the link exit. When the truncated packets arrive at the router, the payload is added to these packets. This way the packet payload only have to traverse the link once. In Figure 2.1 the caching mechanism of CacheCast is depicted. The first packet carries the payload. This payload is stored in a cache on each router. The rest of the packets does not carry the payload, only the header.

The CacheCast mechanism includes three different components; support for CacheCast on the server, a component on the link entry called *Cache Management Unit (CMU)* and a component on the link exit called *Cache Store Unit (CSU)*. The placement of the CMU and the CSU are illustrated in Figure 2.1. The three components can be divided into two groups; server support and network support. These groups are independent of each other, so there is no communication between the server and the components in the network. The details of the server support and network support are explained in the following.

Server support

The use of CacheCast demands that the server is aware of CacheCast support in the network, thus a server component is needed. To be able to take advantage of CacheCast, the same data must be sent to many receivers in a batch. This batching of receivers is done by applications using the CacheCast support.

The CacheCast support for applications is currently offered through the system call msend() implemented in the Linux operating system. The parameters to this system call is a set of sockets and the data to be sent. The set of sockets represents the group of receivers which are receiving the same

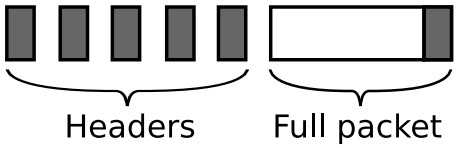

Figure 2.2: Structure of a packet train

data.

To manage the caching of payloads, each CacheCast packet contains a CacheCast header. It contains three fields; payload size (*P_SIZE*), payload id (*P_ID*) and cache index (*INDEX*). The details of these fields are explained in the next paragraph. The msend() system call adds the CacheCast header to each packet and marks the packets as cacheable packets. The packets are marked to let the other CacheCast components identify CacheCast packets. The system call transmits the packets onto the link in a tight sequential order. Only the first packet contains the payload whereas the other packets are truncated, as explained above. This removal of redundant payload makes the CacheCast packets follow a certain pattern. The first packet contains a header and the payload, whereas the rest of the packets only contain the headers. This structure is called a *packet train* and is visualized in Figure 2.2.

The packet train is the source of the performance gains when using CacheCast. The packets are transferred as a batch in a tight sequential order to increase the efficiency of the caching. This batching is done per output link and is a subset of the batching done on the application layer.

Network support

The network support consist of two components; the *Cache Management Unit (CMU)* located at the entry of a link, and the *Cache Store Unit (CSU)* located at the exit of that same link. These components are installed on a per link basis in the network. The main task of the CacheCast network support is to store packet payloads and add and remove payloads from packets. The packet payloads are stored in a cache in the CSU on the link exit. The task of the CMU is to manage this cache and, if the payload of a packet is already stored in the CSU, truncate this packet so it only contains the header. The task of the CSU is to store the payloads and add the payload to packets containing only a header. The routers process the cacheable packets as normal network packets. The packet train structure is only present on the links between the routers. Therefore, CacheCast support in the network can be incrementally deployed from the server. The details of the CMU and CSU are described in the following.

CMU As previously mentioned, the responsibility of the CMU is to remove the redundant payload and manage the cache in the CSU. On the server, a unique payload ID (*P_ID*) is given each payload which identifies, together with the source address, the payload uniquely in

the Internet. All packets in a packet train are supposed to contain a payload with the same *P_ID*.

When the CMU receives a CacheCast packet, its payload is either stored in the cache (in the CSU) or it is not. These events are denoted as a *cache hit* and *cache miss*, respectably. In the event of a cache hit, the CMU removes the payload from the packet, adds the CacheCast header and sends the truncated packet onto to link. In the event of a cache miss, the CMU reserves space for the packet payload in the cache. The CMU contains a table which corresponds to the slots of the cache in the CSU. The *INDEX* field in the CacheCast header identifies where the payload should be stored in the cache. The CMU adds the CacheCast header to the packet and the packet is sent onto the link.

CSU The CSU is the component containing the actual cache. Due to the packet train structure, the CSU can receive two types of CacheCast packets; a packet containing the payload and the header (a full packet) or a packet containing only the header. If a full packet is received, the CSU stores this packet's payload in the cache slot specified by the *INDEX* field in the CacheCast header. If only a header is received, a full packet has already been received and the correct packet payload is stored in the cache. In this case, the CSU attaches this payload to the received header. In both cases, before the packets leave the CSU, the CacheCast header is removed. This is done to enable the router to handle the packets as normal network packets.

2.1.2 Performance gains

The CacheCast mechanism has been developed to increase the performance for single source multiple destination transfer. The performance gains from CacheCast is related to the removal of payloads. The functionality of CacheCast is to remove the redundancy when transmitting the same data over the same link. In this context, the term *redundancy* means the multiple copies of the same packet payload, which is sent over the same link when using many unicast connections. We illustrate this using Figure 2.3. In the figure there are three receivers receiving the same data from a single sender. The data to all receivers has to traverse the first hop link. The figure visualizes the transmission of the same payload (with payload ID 4) to all three receivers using two approaches. (1) The packets are transferred using the standard transmission procedure, i.e. separately on each unicast connection. (2) CacheCast is used to remove redundancy.

When using standard transmission, the data is sent once per connection. Equal packet payloads traverse the link multiple times. When using CacheCast, the duplicate copies of the payload are removed and only the packet headers for the second and third receivers are transmitted. It is clear from the figure how CacheCast can optimize the transmission of the same data to many receivers. When compared to the standard approach, less bandwidth is consumed and the total transmission time to all three receivers is reduced.

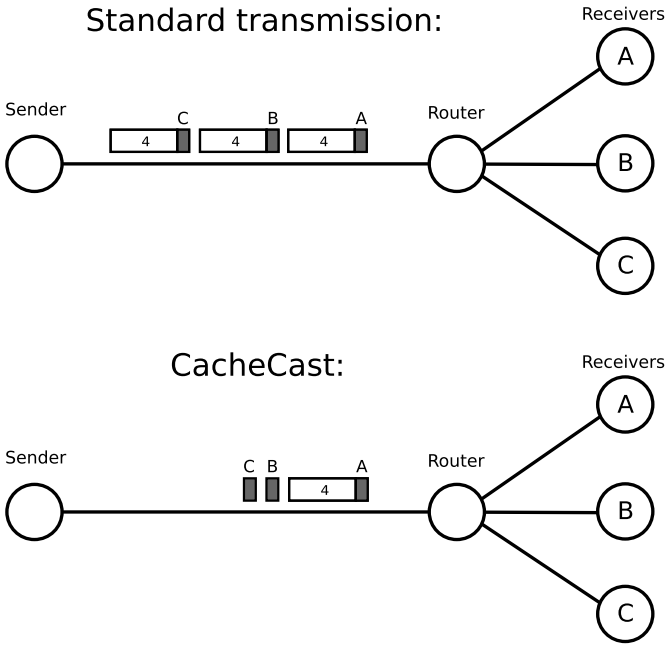

Figure 2.3: Removing redundant payloads with CacheCast

The amount of performance gains that can be achieved by using CacheCast depends on how much redundancy is removed from a link. There are two factors which impact the degree of redundancy removal. The first factor is the size of the payloads. Since, CacheCast removes the redundant payloads, the use of large payload sizes would result in more removed data from the link, when compared to using smaller payload sizes. The second factor is the number of receivers. When the number of receivers increase, the redundancy which can be removed by CacheCast also increase. More specifically, since only the first packet in a packet train carries the payload, when the number of receivers increase, there are more packets which only carries the header. Hence, the performance, when compared to the standard transmission procedure, increases.

2.1.3 CacheCast applications

CacheCast is a link layer mechanism and can be used by any application protocol. However, there are some conditions which needs to be met in order to benefit from CacheCast. Only applications which are able to batch multiple receivers and transfer the same data to all receivers using the msend() function, can gain from using CacheCast. Generally, there are two main groups of applications which deliver content to many receivers: various types of streaming applications and systems offering different kinds of downloadable content. The most obvious group of applications which can benefit from CacheCast are live streaming applications. Such applications transmits the same data to multiple receivers, and the receivers are synchronized *per se*. The amount of modifications needed to add CacheCast support to

an existing live streaming system is minimal. For an implementation and evaluation of CacheCast in the *paraslash*[2] audio streaming system, see [43]. Other types of applications offering a synchronous flow of data to multiple receivers, such as publish/subscribe systems and applications offering live update streams, can achieve the same gains as live streaming applications.

Applications which deliver downloadable content to multiple users can also gain from CacheCast. However, such applications might need more modifications than streaming applications to support the CacheCast mechanism. In this thesis we design, implement, and evaluate such an application.

2.2 Transport protocols

The use of transport protocols is an essential part of any Internet application. In this section we briefly discuss the concept of a transport protocol and introduce the most common transport protocols. These protocols are discussed in order to choose a suitable transport protocol for the CacheCast file server.

Transport protocols reside on the *transport layer* in the TCP/IP model [21], and provide end-to-end communication services for applications in the Internet. Transport protocols is a central part of the Internet Protocol Suite [9] and offer an interface for communication between processes on Internet hosts. Processes communicate using *sockets* which are the endpoints for the inter-process communication. Several transport protocols exist, which have different properties when it comes to reliability, congestion control, data handling, etc. One might divide the different transport protocols into two groups, reliable and unreliable. Reliable transport protocols guarantee that the data is delivered correctly, while unreliable protocols give no guarantees on data delivery at all. In the following we introduce the most common transport protocols in the current Internet.

The *Transmission Control Protocol (TCP)* [37] is a reliable transport protocol and is the most used transport protocol in the Internet. It includes congestion control, reliability and it handles the data as a byte stream. Its reliability is obtained through retransmission of lost segments. TCP is usually used for data transmissions with strict data delivery requirements, such as file transfer, WWW, E-mail etc.

The *User Datagram Protocol (UDP)* [35] is the most used unreliable transport protocol in the Internet. It has neither congestion control nor reliability, and treats the data as individual messages. Due to its simplicity, UDP adds less overhead to the network than TCP and, since it's a connectionless protocol, requires no setup. UDP is mostly used in systems with real-time requirements and where packet loss is accepted, such as streaming systems.

Another reliable protocol, aimed at streaming, is the *Stream Control Transmission Protocol (SCTP)* [45]. Its properties includes reliable, in-order data delivery and congestion control. SCTP is a message-oriented protocol

[2] http://paraslash.systemlinux.org

Table 2.1: Comparison of transport protocols

Protocol	Data handling	Reliable	Congestion control	Preserve message boundary
TCP	Byte stream	Yes	Yes	No
UDP	Message	No	No	Yes
SCTP	Message stream	Yes	Yes	Yes
DCCP	Message	No	Yes	Yes

like UDP, but treats the data as a message stream. Both SCTP and TCP are connection-oriented protocols.

The *Datagram Congestion Control Protocol (DCCP)* [31] is a message-oriented transport protocol which includes congestion control, but no reliability. It offers a TCP-like Congestion Control [19] and a TCP-Friendly Rate Control [20, 24], which both ensure fair bandwidth share in networks with TCP flows. Like UDP, DCCP gives no guaranties on data delivery and is therefore used in systems where data loss is acceptable, or in systems where reliability mechanisms are available on the application layer. DCCP is, like TCP, a connection-oriented protocol. In Table 2.1 we summarize the most important differences between the abovementioned transport protocols.

The CacheCast mechanism requires that the data chunk sent with `msend()` is treated as an individual entity. This is due to the caching and identification of individual packet payloads. Therefore, a transport protocol which preserve message boundaries and handles the data as individual entities are necessary for the CacheCast mechanism. CacheCast currently supports the UDP and the DCCP transport protocols.

2.3 Reliable transfer

The Internet Protocol (IP) [36] on the network layer does not ensure reliable data delivery. The Internet is therefore a network which provides only best-effort transmission of data. This means that data packets may get lost, be delayed, duplicated, reordered or corrupted. When transferring files in the Internet, it is important that the file is correctly received by the receiving host. Therefore, error handling mechanisms are of great importance. In this section we give an introduction to reliability mechanisms which are currently in use in the Internet, and discuss their advantages and disadvantages.

Computer networks are prone to errors. When reliable transfer is necessary, proper actions must be taken in the event of an error. There are several sources of errors in a network, but the end result is either a corrupted packet or a lost packet. The functionality of a reliability mechanism consists of two parts; *identifying* that an error has occurred and *recovering* from it. There are mainly three approaches for identifying errors; adding *checksums* to packets, ordering packets with *sequence numbers*, and adding *timers* to

packet transmissions.

A checksum [10] is extra information added to a packet and it is computed based on the packet contents. The integrity of the packet can be checked by recomputing the checksum and comparing it to the checksum stored in the packet. Checksums are used to identify corrupted packets. All transport protocols explained in Section 2.2 use checksums, in addition to several protocols on lower layers in the network stack.

The second method of identifying errors is using sequence numbers. The rationale of using sequence numbers to identify errors is to ensure that the received packets are ordered correctly. Every packet sent from the sender is tagged with a unique sequence number. When the receiver has ensured that all packets are in the correct order based on the sequence numbers, the receiver has an exact copy of the original data. Sequence numbers can be used to identify lost packets, duplicated packets and reordered packets.

The third error identification method involves using timers. When using timers to identify packet loss, there is an assumption that the packet transmission should take a certain amount of time. Each sent packet has its own timer. The timer value is set either statically or dynamically to a predefined value. If the timer expires, it is assumed that the packet is lost somewhere in the network.

When an error has been identified it must be handled in a suitable way. There are several different mechanisms of fixing errors. On the link layer a *Forward Error Correction (FEC)* [47] scheme is often applied. By using FEC the receiver can both identify and recover from an error by correcting the erroneous data. Usually when a corrupted packet is identified in higher layers in the network stack, it is discarded by the protocol. A discarded packet is essentially the same as a lost packet. Thus, the same recovery techniques is used for both lost and discarded packets.

Basically, there are two different paradigms when it comes to handling packet loss; (1) doing retransmissions of lost packets and (2) to add extra information to the data flow such that retransmissions are unnecessary. We discuss these paradigms in the next two sections.

2.3.1 Retransmissions

The first method recovers from lost packets by retransmitting them. In this method both timers and sequence numbers can be used to identify the lost packets. This method is as follows: If a lost packet has been identified, either by an expired timer or that the sequence numbers of the received packets indicate packet loss, the sender issues a retransmission of the lost packet. This method is implemented in TCP [37]. TCP transmits segments of the original data. The receiver acknowledges the received segments. An unacknowledged segment or an expired retransmission timer indicates loss of data and a retransmission of this segment is issued.

An advantage with the retransmission method is its simple functionality. No heavy computational load is needed on either the sender or receiver. A possible challenge when there are multiple receivers and a single sender, is what is called, a *feedback implosion* [11]. The sender is flooded with

acknowledgements from the receivers which might lead to congestion on the back-channel. In other cases a back-channel might not be available. Additionally, retransmissions of data is not optimal in applications with real-time requirements.

2.3.2 Fountain codes

The other method of handling packet loss is to send extra information so that no retransmission of lost packets is necessary. The most basic approach is to send the original data more than once until the receiver has received all the data successfully. This approach adds too much redundant traffic to the network, so an other mechanism has been developed, called *Fountain codes* [33].

When using the Fountain codes approach the general procedure is as follows: The sender encodes the original data before transmission by creating encoded blocks which are transferred as a stream toward the receiver. The receiver collects encoded blocks and decodes these encoded blocks into the original data.

An original file on the sender's file system can be divided into arbitrary *file blocks*. These file blocks are defined by an encoding schema. An *encoded block* is a combination of several file blocks. When the receiver starts collecting encoded blocks, it can start decoding the original file. For each new encoded block the receiver collects, it is able to reconstruct parts of the original file. The transmission is successful when the receiver has collected enough encoded blocks to decode the whole original file.

Several codes can be used in the Fountain codes approach, for instance Tornado codes, LT codes and Raptor codes [34]. These codes have different properties regarding encoding/decoding complexity, the number of encoded blocks necessary for decoding, etc. All these codes share the common concept that encoded blocks are a combination of some or all of the file blocks. Encoding is the process of combining file blocks into encoded blocks and decoding is the process of extracting the original file from the encoded blocks. The encoded blocks are linear combinations of file blocks, which can be represented as linear equations. The variables in the equations represent the file blocks and the coefficients are chosen based on the encoding (e.g. randomly or from a generator matrix). Decoding is equivalent to solving a system of linear equations represented by the encoded blocks that has been received. The decoding can be done iteratively by elimination of variables. When all variables have been eliminated, the system of equations is solved and the original file has been reconstructed. As an example, the following three equations represent the encoded blocks e_1, e_2 and e_3. The variables x, y and z are the file blocks.

$$\begin{aligned} 3x + 2y - z &= e_1 \\ 2x - 2y + 4z &= e_2 \\ -2x + y - 2z &= e_3 \end{aligned}$$

The solution to this system of equations is

$$\begin{aligned} x &= -\tfrac{1}{2}e_2 - e_3 \\ y &= \tfrac{2}{3}(e_1 + 2e_2 + \tfrac{7}{2}e_3) \\ z &= \tfrac{1}{3}(e_1 + \tfrac{7}{2}e_2 + 5e_3) \end{aligned}$$

which can be solved using row reduction on the coefficient matrix. Therefore, the decoding procedure is basically to solve a system of equations by applying row reductions iteratively on the coefficient matrix.

The ideal coding for Fountain codes would encode N original file blocks into N encoded packets. To achieve this, every encoded block must contain enough information to eliminate variables, and the equations represented by the encoded blocks must be linearly independent. In the previous example, the three equations are linearly independent. Since there are three variables, all three equations are necessary to solve the system of equations. This corresponds to three file blocks being encoded into three encoded blocks. However, this ideal coding has proved difficult, but good approximations exists [34]. In practice, N file blocks are encoded into $N + \delta$ encoded blocks, where δ is a small percentage of extra encoded blocks.

The major advantage of using Fountain codes is that packet loss is not an issue. The receiver just collects enough packets to be able to reconstruct the original data. If an encoded block is lost during transmission, the receiver will just have to wait for the next encoded block to arrive. This is possible since the equations, represented by the encoded blocks, are linearly independent. Every encoded block the client receives can be used to decode the original data. A consequence of this is that no signaling or retransmissions are necessary.

The main disadvantage with Fountain codes is the computational load on the sender and receiver, due to the encoding and the decoding procedures. However, it has been shown by Shojania and Li that the encoding and decoding can be performed with satisfying performance even with moderate hardware specifications [42].

2.4 Content popularity and request rate

A huge collection of various kinds of content, such as web sites, videos, images, applications etc., are available through the Internet. Every second, a large amount of new data is distributed from all over the globe. The majority of the content is published on the web for others to consume. In this section we investigate how users consume content, and show that users often request popular content in bursts.

An important property of content on the Internet, is the popularity. In this context we define the term *popularity* as a measure of user interest in some content. For example, a web site with many users is a popular web site. The popularity of content varies based on demand. The *request rate* is a measure of the number of user accesses over a specified time interval. The request rate of content depends mostly on the type of content, but it also

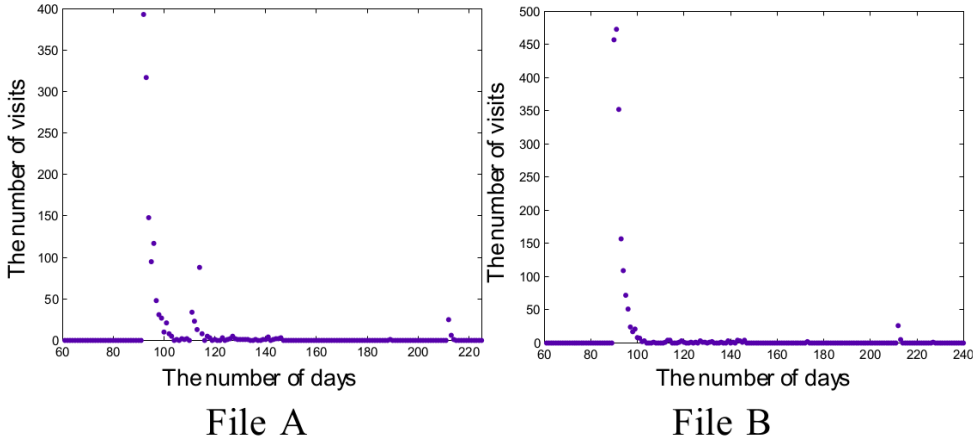

Figure 2.4: User requests along time for two files [12]

depends on when the content is made available and when users are ready to consume it. For instance, the request rate often varies with the hours of the day. During weekdays there are peeks in user accesses around noon (12.00 - 14.00) and after work hours (18.00 - 21.00) [46].

The distribution of user requests to content distribution systems is often modeled using a Zipf distribution or Poisson distribution. In a Poisson distribution, the request rate quickly grows until it reaches a maximum and then it decreases more slowly than the increase. Chang et al. measured the request rate of files in a media server over an 8-month period. His findings were that "user access behavior is bursty both in general and per-file" [12]. For the two most popular files, 78.8% and 91.36% of the user accesses were experienced during a time interval of a few days within the 8-month period. These observations can be seen in Figure 2.4. The highest peeks in the user accesses for both files is from approximately day 90 - day 100. Chang et al. also noticed that, for the less popular files, no specific access pattern could be discovered.

The phenomenon of many users requesting some content within a short time interval is often called a *flash crowd*. Such a user behavior can often be seen in temporary content, like news stories. Jung et al. lists some widely known examples such as "the release of Ken Starr's report on a few Web sites in 1999, popular webcasts like that of Victoria's Secret company, and sports events like the Olympics" [29]. In these examples, information about the events is known in advance, but flash crowds can originate without warning. A typical example is when www.cnn.com became unavailable due to a high increase of requests on September 11, 2001.

The lifetime of a certain content could be either short lived or long lived. Typical short lived content is news stories and temporary web sites for special events, while typical long lived content is downloadable movies and program installation files. The examples in the previous paragraph are short lived content. As an example of long lived content we present the download statistics of two applications, namely *Apache OpenOffice* and

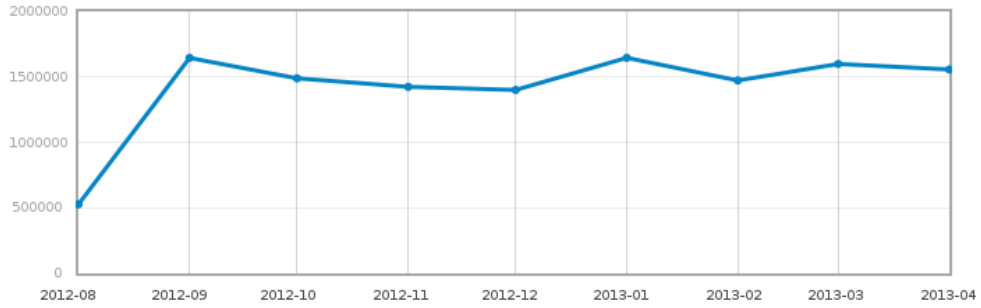

Figure 2.5: Download statistics for OpenOffice [2]

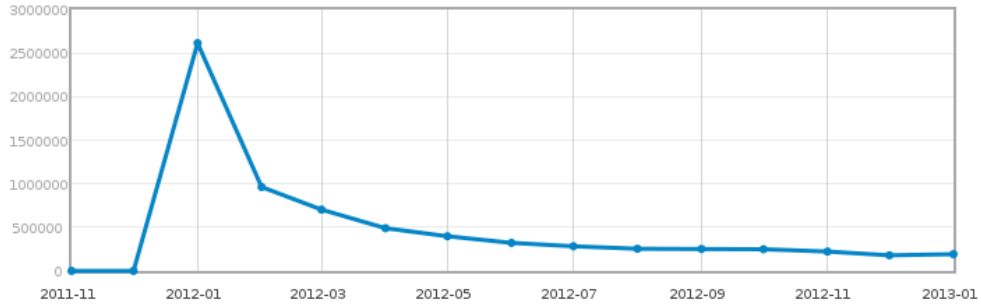

Figure 2.6: Download statistics for Notepad++ Plugin Manager [3]

Notepad++ Plugin Manager. These two applications have different user access patterns. The download statistics of *OpenOffice* is depicted in Figure 2.5. It reveals a stable download rate of around 1.5 million downloads per month. This does not follow the bursty behavior explained above. The statistics for *NotePad++ Plugin Manager* is depicted in Figure 2.6. Here we see a different behavior. Over 2.6 million requests are registered within the first month. However, the number of requests quickly drops over 50% in the second month and continues to drop slowly for each month. From these two examples and from the findings of Chang et al., it is clear that the user access pattern can vary even for similar types of content. Still, for popular content, the bursty behavior of user accesses is clearly present.

When new content is introduced on the Internet, the popularity is affected. Yu et al. states that "the availability of new content captures users' attention and requests, thereby changing the distribution of user requests" [49]. Such an introduction could be either when a certain content is made available or when users are starting to draw attention to it. For example if a link to a web site is promoted on social media by an influential person, the popularity of the web site quickly grows.

2.5 The ns-3 network simulator

The CacheCast file server has been implemented in the ns-3 network simulator. Here we give a brief introduction to the core elements of this simulator.

The ns-3 network simulator [6] is a discrete-event simulator, which is used primarily for scientific and educational purposes. Ns-3 is built from scratch without any connections to its predecessor ns-2 [5]. Both the simulator core and the user defined simulation scripts are written in C++. The simulator is built as a library. The simulation scripts are regular C++ applications which import and use the ns-3 simulator library.

The components of ns-3 are abstractions of the components found in real-world networks. These components are designed to closely resemble the functionality of current network devices and protocols. The following list contains a description of the most essential components in ns-3. The caption of each description represent both the general name of the component and the C++ class name used in ns-3.

Application In a computer there are generally an operating system and user applications which runs on top of this operating system. In ns-3 there is no notion of an operating system. However, there is the notion of an application. In ns-3, an Application is the driving force, which generates activity for the simulator. This activity is basically sending and receiving network packets. Applications reside on Nodes.

Node A Node in ns-3 is the abstraction of a general computing device. It resembles what is often called a host in Internet jargon. Both end-hosts and routers are represented as Nodes in ns-3. Just as in a real computer, different elements can be added to a Node, like Applications, network stacks, NetDevices, etc.

NetDevice In order for a computer to transfer network packets, a hardware device called *network card* must be installed in the computer and software support for this network card must be available through the operating system. In ns-3, both the software support and the hardware abstraction is covered within a NetDevice. A NetDevice is installed on a Node and it transmits network packets over Channels.

Channel In a real network, computers are connected using for instance wired links or Wi-Fi channels. In ns-3 the connection between different computers is represented by a Channel, i.e. Channels are the elements which enable communication between Nodes.

Socket A Socket in ns-3 is the endpoint of a communication flow across the network and is located on the Transport layer. Applications sending or receiving data, communicates through a Socket. Packets are sent and received with the `Socket::Send` and `Socket::Recv()` functions, respectably.

Packet A Packet in ns-3 is an abstraction of the data unit which is being transferred through the simulated network. Packets are created by

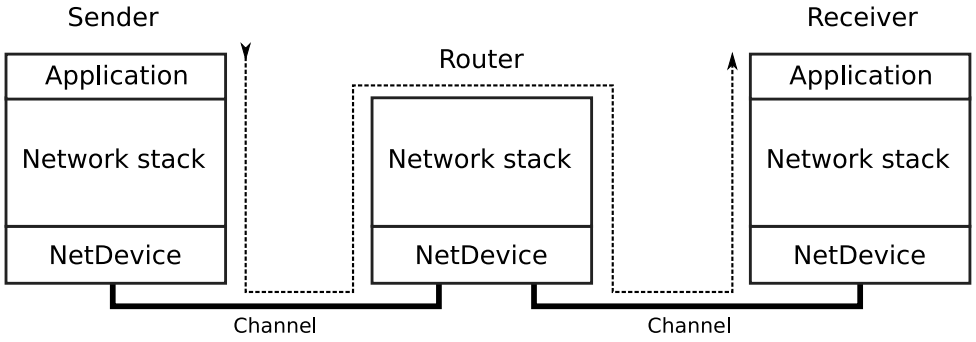

Figure 2.7: Overview of the basic components in ns-3

Applications and are sent using Sockets. Headers are added to Packets on different layers in the network stack to support various network protocols.

In Figure 2.7 there is an overview of the basic components of a simulated network in ns-3 and how they are related. The three Nodes are connected by two Channels. Each end of a Channel is connected to a NetDevice and the NetDevices are installed on the Nodes. On all Nodes a network stack is present, and on the two end-hosts Applications are installed on top of the network stack. The dashed line indicates the data flow from the sender to the receiver.

Chapter 3

Basic design considerations

The CacheCast file server requires a specific design in order to benefit from the CacheCast mechanism. As briefly noted in the previous chapter, the effort to include support for CacheCast in existing file transmission systems is larger than for live streaming systems. This is due to the nature of file transfer and the CacheCast mechanism. The CacheCast mechanism is built for single source multiple destination transfer and requires that the same data is transferred to multiple receivers. Most standard file transfer systems are built to support single receivers downloading from a single server. The CacheCast file server must be designed to transmit the same data to multiple receivers, in order to gain from CacheCast.

In this chapter we discuss some general design considerations for a file server with CacheCast support. We explain why a custom file server design and architecture is necessary and give a basic description of the functionality needed to enable CacheCast support. We start by giving an introduction to file transmission in general.

3.1 General overview of file transmission

From a user's perspective, file transmission can take many forms. When browsing the World Wide Web (WWW), files, such as images, HTML files, CSS files, etc., are transferred by the HTTP protocol. If a user receives an e-mail containing an attachment, both the e-mail text and the file attachment has to be transferred to the user's computer to view their contents. When a user updates his operating system, the update procedure requires that many new files are transferred from the update server. Also, a user can explicitly connect to a file server, choose the file he wants, and start the download procedure.

In all of these scenarios the file transmission procedures are almost identical. There is a single computer acting as a *file server* and a computer acting as a *client*. The server contains a list of files which can be downloaded. A client connects to the server, selects which file to download, and starts the file transmission. The server transmits the file to the client.

On a high level, a file server has two main components; *File selection* and *File transmission*. These components are visualized in Figure 3.1. The file

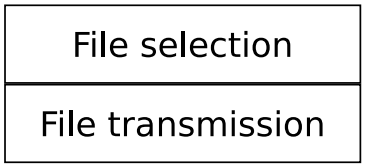

Figure 3.1: Basic file server architecture

selection is, in most cases, initiated by a user, while the file transmission is the underlying functionality which transfers the file from the server to the client (the user's computer). We illustrate this basic file server functionality by an example.

A user wishes to download the latest version of his favorite Linux distribution. The user has browsed to the web site where he can click on a link to start the download. After the user has clicked on the link, the web browser sends a request for the file to a file server. The file server receives the request and prepares to transfer the requested file to the user's computer. The *file selection* has now been performed and the *file transmission* is started. The file is now being transferred through the network to the user. At some point in time the whole file has been transferred and the transmission procedure ends. The user is now able to consume the contents of the file.

In the current Internet there are many highly used protocols which includes similar file transmission functionality. Examples include *Hypertext Transfer Protocol (HTTP)* [17] for web content, *Simple Mail Transfer Protocol (SMTP)* [30] for e-mail transfers and *File Transfer Protocol (FTP)* [38] for general file transfers. All these protocols use TCP to ensure reliable transfer. The procedure for transferring a file using these protocols is very similar. Throughout this thesis we use FTP as the reference protocol. A simplified description of the functionality of FTP is as follows: A client connects to the server by creating a new TCP connection. The client selects which file to download and tells the server to initiate the file transmission. The server starts forwarding the contents of the file to the client over the TCP connection. The default transfer mode in FTP is *Stream mode* [38], and when this is enabled FTP sends the data as a sequential stream of data. In the rest of this thesis we assume that *Stream mode* is enabled. TCP ensures that the file is transferred correctly by dividing the data into segments, assigning sequence numbers to these segments and issuing retransmissions when segments are lost in the network. By using the sequence numbers, TCP assures that all received segments are correctly ordered. TCP also adjusts the transmission rate to the client's available bandwidth. In Figure 3.2 the architecture of FTP is depicted. The FTP commands and the data connection is separated. On both sides of the *Data Connection* there is a *Data Transfer Process (DTP)* which established and manages the Data Connection. The Server and Client *Protocol Interpreter (PI)* handles the user commands and starts the file transmission over the Data Connection. In FTP every aspect of

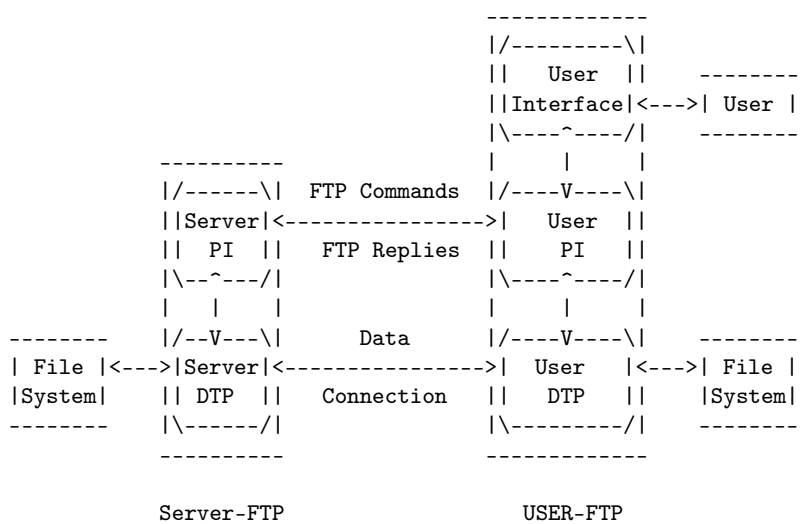

Figure 3.2: FTP architecture [38]

the actual file transmission is handled by TCP on the Transport layer. The file selection (and other user commands) is performed on the Application layer.

In this thesis our main focus is the design of the file transmission part of the file server. The file selection (and other user commands) is beyond the scope of this work. We present the design of a CacheCast file server which is a general file transmission mechanism that can be used by any application layer protocol.

3.2 Multiple clients

An FTP server is a multi user system, i.e. it has support for multiple clients downloading files concurrently. A client can connect to the server at any time and request any file. FTP is designed for single source, single destination transfer. Thus, for each client connected to the server there is one TCP connection. Each client is served separately on each unicast connection.

When multiple clients are downloading the same file at approximately the same time, there are overlapping time periods between the download procedures. During these periods the same file is transferred to multiple clients at the same time. When the request rate to a file server is high, as in the event of a *flash crowd* (cf. Section 2.4), many clients will connect to the server within a small time interval, creating multiple overlapping time periods between the clients.

In Figure 3.3 two clients are downloading the same file. The rectangles indicate the time frame needed to download the file. When the second client starts downloading, the first client is not yet finished. This adds an overlapping time period between the two downloads - visualized in gray in the figure. Within this overlapping time period Client 1 and 2

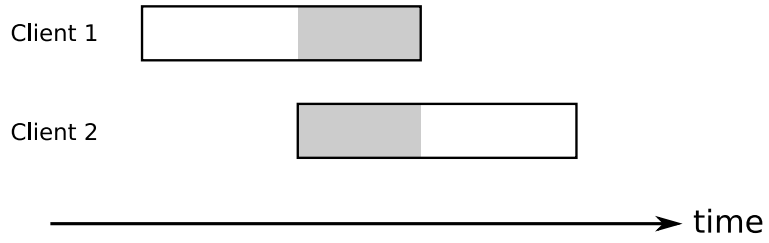

Figure 3.3: Two downloads overlapping

are downloading the same file concurrently. However, they are receiving different parts of the file. Client 1 is receiving the end of the file, while Client 2 is receiving the beginning of the file. This is an important insight for the discussion in the next section.

Within overlapping time periods, CacheCast can be used to optimize the transmission. The specific techniques to enable this optimization is explained in the following sections.

3.3 Requirements

When transferring files through a network there are two requirements that need to be fulfilled. First, the whole file must be correctly received by the client, i.e. without any corruption. Second, the client's download speed should correspond to its available bandwidth capacity. The first requirement is concerned with reliability, while the second requirement ensures satisfying performance. If the first requirement is not fulfilled the download procedure would be worthless, since the file would be corrupted. If the second requirement is not fulfilled, the data would still be successfully transferred, but with a slower transmission rate. The download time experienced by the client would be less than the optimal download time. Therefore, it is important to fulfill also the second requirement, even though it is not as strict as the first requirement.

As discussed in Section 3.1, in a standard FTP server both of these requirements are handled by TCP. Retransmission of erroneous data ensures that the whole file is successfully received, and TCP's congestion control adjusts the transmission rate to match the available bandwidth. TCP also ensures fair share of bandwidth resources between concurrent downloads.

To optimize the transmission to multiple clients, we use the CacheCast mechanism. In Section 2.1 we explained the functionality of the CacheCast technique. In order for CacheCast to be beneficial, it adds a third requirement to the file server: The file server must be able to transmit the same data to multiple clients within a small time window. If this requirement is not met, the CacheCast mechanism can not remove redundant data.

The three requirements above, must be fulfilled in order to build a CacheCast file server. The first requirement must be handled by a reliability

mechanism, which ensures that the data is transferred correctly to the client. For the second requirement, the transmission rate must be adjusted to match the client's available bandwidth. These first two requirements can be fulfilled solely using specific techniques and algorithms, cf. the functionality of TCP.

The third requirement specifies that the file server must be able to transmit the same data to multiple clients synchronously. In this thesis we define the term *synchronous transmission* as the procedure of transferring the same data chunk to multiple clients within a small time window. A fundamental part of the CacheCast mechanism is the packet train structure. This structure is a result of CacheCast removing redundant packet payloads and transmitting the packets onto the link in a tight packet train. The packets must be in a tight sequence in order to improve the efficiency of the CacheCast caching mechanism. Therefore, synchronous transmission is necessary for CacheCast to create this tight packet train. Thus, to utilize the CacheCast mechanism, the CacheCast file server must ensure synchronous transmission of the data.

An essential condition for utilizing CacheCast is that there are multiple clients downloading the same file concurrently. For instance, if a file is only requested occasionally, there will be no overlaps, and no optimization can be achieved by CacheCast. However, in Section 2.4 we explored the popularity of files and observed that certain files often are requested by many clients over a short time period. This fact can ensure that synchronize transmission to multiple clients are possible.

3.4 Synchronous transmission

When a large file is transferred through a network, it is divided into smaller chunks, also called blocks. The common wisdom is that these file blocks are transferred sequentially from the server to the client. This approach is implemented in TCP. However, sequential transmission order introduces problems when synchronous transmission of blocks is necessary. In the following discussion we assume a sequential transmission of blocks, in order to analyse the problems and consequences of this ordering.

3.4.1 Challenges with synchronous transmission

In order to achieve synchronous transmission, there are mainly two challenges; (1) variable bandwidth capacities of the clients and (2) different arrival times to the server. We investigate these issues in the following.

The first challenge is related to variable bandwidth capacity among clients. In the Internet, the bandwidth of each host varies greatly. From a content distribution perspective, this variation needs to be handled. A client with a low bandwidth capacity is not able to receive data as fast as a client with a high bandwidth capacity. This will affect how the sender treats these clients. We illustrate this issue with an example. In Figure 3.4 there are two clients with different bandwidth capacity, downloading the same file. The rectangles in the figure represent the transmission of blocks over time. The

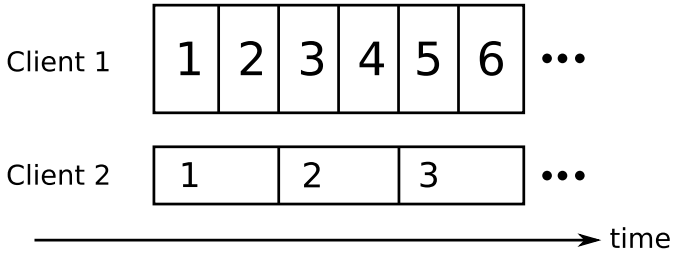

Figure 3.4: Clients with variable bandwidth capacity

width of a rectangle indicates the amount of time to transfer the block and the height indicates the bandwidth capacity of the client. Client 2 has half of the capacity as Client 1. Both clients start the download procedure at the same time and the transmission of the blocks is sequential, starting at block 1. The first block is transferred to both clients synchronously. However, due to the fact that Client 1 downloads twice as fast as Client 2, synchronous transmission is no longer possible after the first block has been transferred. Since sequential transmission order is assumed, the file server is not able to transmit the same block synchronously to both clients. The solution to this problem in a streaming environment is to offer the clients streams with different rates. The clients can choose the rate that suits them best.

The second challenge is related to the arrival times to the server. In a file server, the clients might connect and request a file at any point in time. While the clients in a live streaming scenario are synchronized *ipso facto*, the clients in a file server is not. Thus, a file server must take into account that clients can request a file at different times. There are, however, valuable predictions on how users access content (see Section 2.4). To illustrate the problem, we present in Figure 3.5 an example of clients arriving at different times to a file server. Six clients connect to the server and request the same file, but at different points in time. In this example all clients have the same bandwidth capacity. The rectangles in the figure corresponding to each client, represents the time frame in which the clients are downloading the file. For example the download for Client 1 starts at time t_1 and ends at t_2. As in Figure 3.3 the gray areas indicate the overlapping between the downloads.

From this example we can draw some conclusions. From time t_1 to t_2, no synchronization is possible since there is only one connected client. Client 2 and 3 both arrive at exactly t_3 and are therefore synchronized in time. As for Client 4, 5 and 6 there are overlaps in the download time frame, but they are not synchronized in time. With sequential transmission order the server can only synchronize the transmission to Client 2 and 3.

The arrival times in a standard FTP server is not an issue since the clients don't need to be synchronized. In FTP, the issue of variable bandwidth capacity is handled by TCP. For each TCP connection, the transmission rate is adjusted to the client's available bandwidth capacity.

In addition to the two challenges above, there is a third issue which complicates synchronous transmission. This is the issue of packet loss in

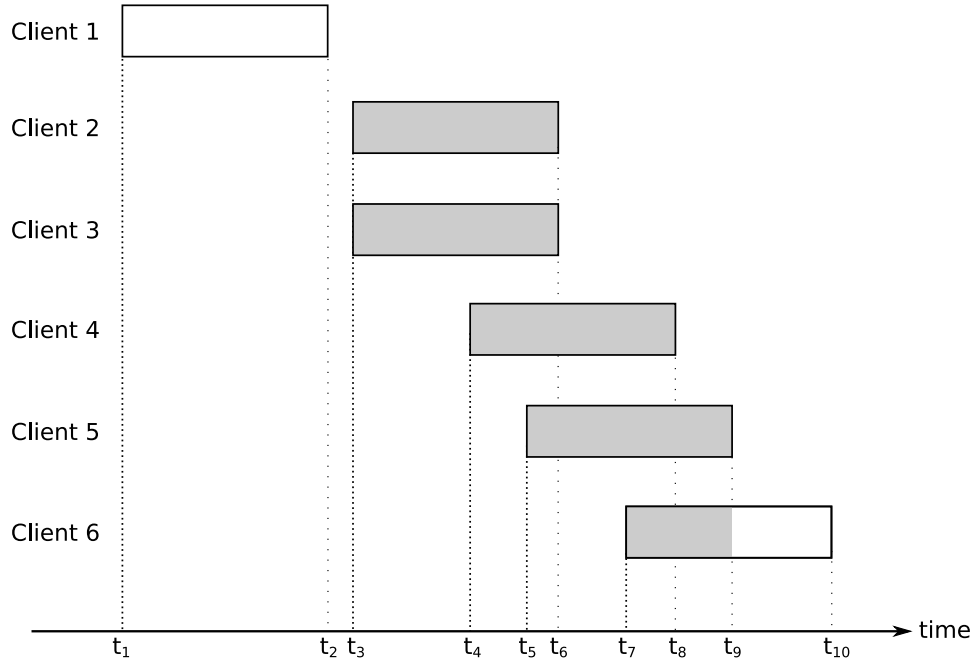

Figure 3.5: Example of multiple clients, with variable arrival times to the server, downloading the same file

the network. Whenever a packet is lost, the client does not receive all the data. To ensure reliable transfer, such packet loss must be fixed. In order to fix packet loss the server must transmit more data to the client. This transmission of extra data should also be synchronous.

To summarize, the challenges with synchronous transmission are related to the variable bandwidth capacities among the clients and the arrival times to the server, and packet loss complicates the matter further.

3.4.2 Approaches for achieving synchronous transmission

So far in this section we have assumed a sequential transmission of blocks. However, such a sequential transmission order is not required. In order for a file transmission to be successful, the transferred file should be an exact copy of the original file, i.e. it should contain a copy of all the original blocks and they should be ordered as in the original file. Based on this insight, it is clear that as long as the client is able to restore the original block order, the blocks can be transferred in any order from the server. The server must provide enough information so the client knows the correct ordering. As in TCP, sequence numbers are used to specify the correct order of the blocks. Therefore, as long as all the blocks are received by the client, the reordering can be done on the basis of the sequence numbers. The ability of transferring the blocks in any order is the basis for the functionality of the CacheCast file server.

Based on the previous discussion we present three approaches for enabling synchronous transmission. In the first two approaches we assume a sequential transmission of the blocks, as in TCP, but in the third approach this assumption is relaxed.

1. Clients have to connect to the server at exactly the same time (as for Client 2 and 3 in Figure 3.5). Then the server could transmit the data sequentially to each client using CacheCast.

2. The server defines multiple start times for the file transmission. Clients arriving between the start times must wait until the next start time, i.e. the clients are batched into synchronized groups. Dan et al. proposed a similar solution for a Video-on-Demand server [14].

3. The server arranges the transmission of the blocks within the overlapping periods in such a way that the synchronous transmission is achieved. The transmission order is not sequential, it is chosen by the server to enable synchronous transmission. The functionality of this approach is not constrained by neither the clients' arrival time nor the bandwidth capacity.

The first approach would give the highest performance gains, since the benefits from CacheCast would be similar to the benefits in a streaming system (cf. Section 2.1.3). The biggest drawback with this approach is that it is very unlikely that multiple clients will start the download at exactly the same point in time. This approach is therefore, in all practical sense, unusable.

The solution in the second approach overcomes the problem of the first. The clients do not need to start the download at the same time. The inconvenience with this approach is that clients will have to wait for the download procedure to start, which will decrease the overall performance. Both the first and the second approach do not handle the issue of variable bandwidth capacity. This could be handled by offering streams with different rates, but this would also decrease the performance.

The assumption in the first and second approach is sequential transmission order of the blocks. An effect of this assumption is that the download procedure for the clients must be started all together. This makes it difficult to synchronize clients arriving to the server at different times. As a consequence, the third approach does not require a sequential block order. The blocks can safely be reordered before transmission, as argued about in Section 3.4. The third approach overcomes the issues with both the first and the second approach, since the clients' arrival time and bandwidth capacity does not influence the functionality of this solution. The synchronous transmission requirement is therefore meet without taking the arrival times and bandwidth capacity into consideration. To enable this third approach, the server must do some extra work to arrange the transmissions in a suitable way. In this thesis we investigate, implement and evaluate the third approach.

3.5 Optimizing the server transmission using CacheCast

In the previous section we explained that some special arrangement of the blocks is needed to benefit from CacheCast. The core of this arrangement is the server's ability of transferring the blocks in any order. With this ability, the transmission can be optimized using CacheCast.

In Section 3.3 we discussed that synchronous transmission is necessary to utilize the CacheCast mechanism. We defined synchronous transmission as the procedure of transferring the same block to multiple clients within a small time window. From this definition it is clear that in order to benefit from CacheCast, the server must transmit the same block to the overlapping clients. Thus, the server must reorder the blocks so that the same block is transferred in the same time frame to the overlapping clients. To illustrate this reordering we present in Figure 3.6 and 3.7 two examples of the transmission of a file to two clients, with a different order of the blocks. In the figures, the numbers indicate which block is sent during a certain time interval. As we can see, during the overlapping time intervals, the server arranges the blocks so that the same block is sent to both clients at the same time. In Figure 3.6 the block order is continuous, while in Figure 3.7 the order does not follow a specific pattern. In both cases, when the transmission is finished, all the blocks of the file are received, but in different order. When the clients have reordered the blocks correctly, the transferred file is an exact copy of the original file.

These two examples highlight how the reordering of blocks overcomes the challenge of different arrival times to the server explained in the previous section. As long as there are overlapping time periods between the downloads, the CacheCast file server is able to optimize the transmission by reordering the blocks.

The other major challenge from the previous section is variable bandwidth capacity among the clients. The solution to overcome this issue is also to reorder the blocks. In Figure 3.4 we gave an example of two clients with different bandwidth capacities, downloading the same file. We discussed how sequential order of the blocks prevents synchronous transmission. In Figure 3.8 we modify this example in order to present how block reordering can enable synchronous transmission. The block order for Client 1 is the same as in the previous example. The block order for Client 2 is, however, modified such that the same block is transferred to both clients synchronously. CacheCast is then able to optimize the transmission of block 1, 3 and 5 even though the overlapping clients have different bandwidth capacities. Still, Client 1 receives the data twice as fast as Client 2. As for block 2, 4 and 6, which is not transferred to Client 2 in sequence, they must be transferred at a later time. Such issues where blocks are not transferred because of different bandwidth capacities, is handled by a reliability mechanism (see Section 3.7). We introduced packet loss as a third challenge to achieve synchronous transmission. However, packet loss is handled in the same way as blocks which have not been transferred. The reliability mechanism notices that blocks are either not sent or lost, and issues a transmission of these blocks at a later time.

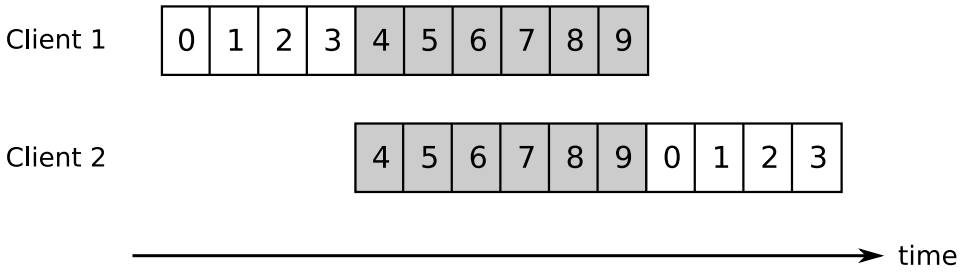

Figure 3.6: Transmission of the same blocks continuously during the overlapping time periods

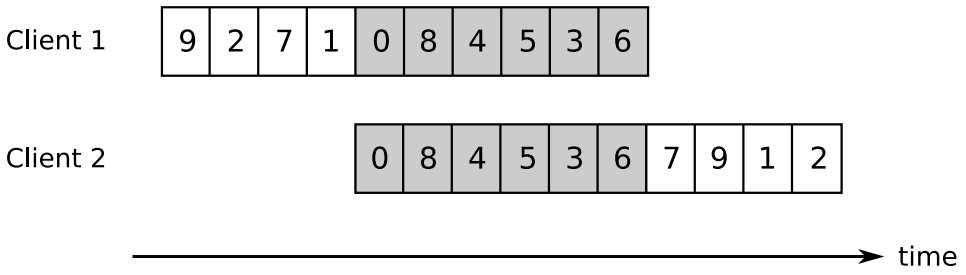

Figure 3.7: Transmission of the same blocks randomly during the overlapping time periods

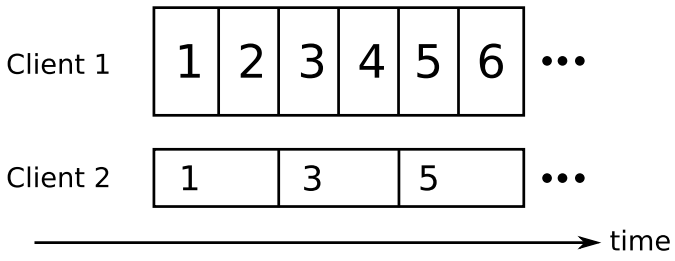

Figure 3.8: Synchronous transmission to clients with variable bandwidth capacity

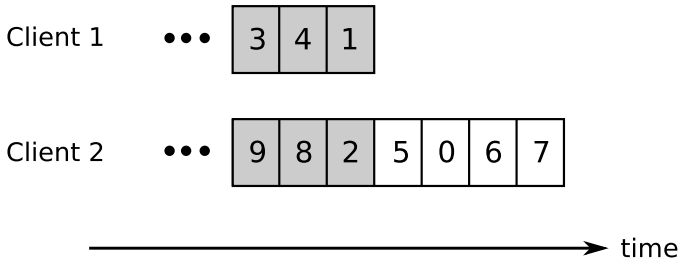

Figure 3.9: Example where synchronous transmission is not possible within the overlapping time period

In all of previous examples, the server is able to reorder the blocks in the overlapping time periods such that synchronous transmission is achieved. However, this may not always be possible. In Figure 3.9 we give an example of an overlapping time period where no synchronous transmission is possible. The two clients in the figure are at the end of the file transmission procedure and are both lacking a set of blocks. However, as we see from the figure, these two sets of blocks are disjoint. Therefore, during the overlapping time period no common block can be transferred to both clients, and no optimization can be achieved.

The problem in this example is that the two clients are not missing a common block. This means that while synchronous transmission can be achieved in most cases during the overlapping intervals (cf. Figure 3.6 and Figure 3.7), it can not be assured (as this example shows). Hence, transmission of the original blocks together with reordering, is not enough to enable synchronous transmission in all cases. To overcome this limitation we introduce the concept of coding.

In Section 3.4 we explained how a large file is split into smaller parts, called blocks, before transmission. Each block contains a part of the original file. When all the blocks are put together in the correct order, a copy of the original file is constructed. However, the blocks do not need to contain the exact original data, as long as it is possible to reconstruct the original file. In Section 2.3.2 we explained the functionality of Fountain codes. To restate, the general functionality of Fountain codes is to encode the file blocks before transmission, and transmit a stream of encoded blocks toward the clients. The original file is divided into arbitrary file blocks and each encoded block is a combination of several of these file blocks. The client must decode the blocks in order to reproduce the original file. The file is successfully transferred when the client has received enough encoded blocks so that it can decode the whole original file. As explained in Section 2.3.2, an important property of Fountain codes is that whichever encoded block the client receives, it can be used in the decoding process. This property makes Fountain codes appropriate for synchronous transmission. When using Fountain codes, the problem of clients not missing a common block is no longer present. Each client can now receive (and use in the decoding

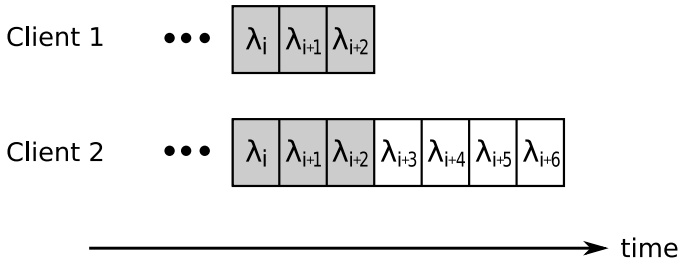

Figure 3.10: Example where synchronous transmission is achieved in the overlapping time period by using Fountain codes

process) each block the server encodes and transmits. Fountain codes is therefore the solution to the problem in Figure 3.9. We modify the previous example to use Fountain codes and depict this in Figure 3.10. Here the j'th encoded block is indicated as λ_j. In the overlapping time frame synchronous transmission is achieved, since the same encoded block is transferred to both clients.

To summarize, a file is divided into file blocks. These blocks can either be transferred in its original form, or new encoded blocks can be created based on the file blocks. When transferring the original blocks it may happen that synchronous transmission can not be achieved. When using Fountain codes, synchronous transmission can always be achieved, but at the cost of the encoding and decoding process. In this thesis we explore both approaches.

3.6 General server-side transmission procedures

In the previous section we explained how the transmission can be optimized using CacheCast. We also presented some examples of optimizing the transmission. In order to define an algorithm to enable synchronous transmission, the examples need to be generalized into a general procedure.

We have discussed two approaches for block transmission; transmitting the original file blocks or encoding the file blocks into encoded blocks before transmission. Therefore, in the following subsections, we present two general procedures, namely *Transmission of original blocks* and *Transmission of encoded blocks*.

3.6.1 Transmission of original blocks

In order to transfer original blocks synchronously, the order of blocks must be altered. The main task when transferring original blocks is therefore to reorder the blocks, such that the same block can be transferred to multiple clients synchronously. Since new clients can arrive and connected clients can leave at any point in time, the reordering must be done based on the current state of the clients. This means that which block to transmit, is selected just before the transmission procedure. An algorithm controls the block

selection. The are many possible algorithms which could be used. Here we present two such algorithms:

Round Robbin In this algorithm the blocks are selected sequentially. When the first download starts, an index pointer is set to the first block of the file (block 0). The index pointer is incremented when a new block has be transferred. Whenever a new client enters, it starts receiving the block on which the index pointer currently points. When the last block of the file is transferred, the index pointer is set to block 0 again. In the previous example in Figure 3.6 this algorithm is used. When Client 2 starts downloading, the current block is block number 4, so it starts receiving this block. When Client 1 is finished downloading, the index pointer wraps around and Client 2 starts receiving the blocks from the beginning of the file.

Most Wanted Block In this algorithm the block selected for transmission is the one most clients are missing. Each block reside in a priority queue where the priority is the number of clients missing this block. This algorithm requires more computation than Round Robbin, in order to identify which block to transfer.

In both algorithms the server must know which block each client has received, so this information must be stored on the server. In this thesis we use the Round Robbin algorithm, because of its simple functionality. In Algorithm 3.1, pseudo code for the transmission procedure using Round Robbin is presented. The index pointer is called *blockIndex* and *blockCount* is the total number of blocks in the file.

Algorithm 3.1: Transmission of original blocks using Round Robbin

```
1 blockIndex ← 0;
2 while there are clients downloading do
3     block ← file block number blockIndex;
4     clientSet ← set of clients missing block;
5     transmit block to clientSet;
6     blockIndex ← (blockIndex + 1) % blockCount;
7 end
```

3.6.2 Transmission of encoded blocks

The procedure of transmitting encoded blocks uses the Fountain codes approach. Therefore, both the terms *transmission of encoded blocks* and *Fountain codes* is used interchangeably. This procedure contains two major steps: (1) Encode a new block for transmission and (2) transmit this block to all clients. Pseudo code for this procedure is available in Algorithm 3.2. Compared to transmission of original blocks there are two major advantages. First, the server does not need to keep track of which blocks each client has received. Second, the encoded block can be transferred to all clients, not

only the ones missing a certain block. This is an important insight for later discussion.

Preparing a block for transmission requires more computation than Round Robbin, since the block has to be encoded. However, the encoding process could run in parallel with the transmission process, or the encoded blocks could be encoded in advance.

Algorithm 3.2: Transmission using Fountain codes

```
1 while there are clients downloading do
2     encodedBlock ← encode a new block;
3     transmit encodedBlock to all clients;
4 end
```

In Table 3.1 a summary of the advantages and disadvantages of transmission of original blocks and Fountain codes is presented.

3.6.3 Optimization effects

To conclude this section about CacheCast optimization we discuss the effects and consequences of the optimization. The CacheCast technique removes redundant payloads of packets traversing the links in the network. The effect of this payload removal is that less bandwidth is consumed in the network. A consequence of the payload removal and the tight packet train structure is that the data traverses the network faster than when using standard file transmission.

To illustrate how the benefits of CacheCast vary, we expand in Figure 3.11 the example from Figure 3.5 by indicating the multiple overlapping periods. The rectangles are colored based on the number of clients downloading the file concurrently. The darker the color, the higher is the number of concurrent clients. For example from time interval t_4 to t_5, there are three clients downloading the file. When multiple downloads are overlapping, the CacheCast file server is able to synchronize the transmission to all overlapping clients. As discussed in Section 2.1 the performance of

Table 3.1: Transmission of original blocks vs. Fountain codes

	Transmission of original blocks	Fountain codes
Pros	No modification of blocks.	All clients can receive all transmitted blocks. Information about which blocks the clients have received is not necessary.
Cons	Only the clients missing a block receives it. Record of received blocks must be stored per client on the server.	Extra computation involved in encoding/decoding.

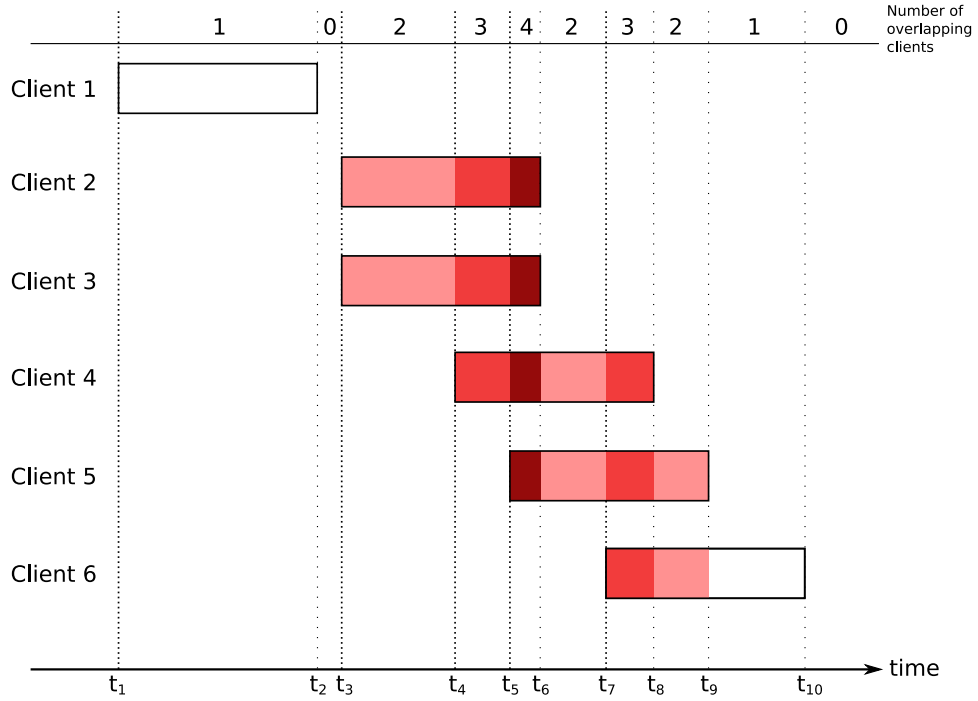

Figure 3.11: Example of multiple clients downloading the same file during multiple overlapping periods

CacheCast increases with the number of receivers. In time interval $[t_5, t_6]$ there are four overlapping downloads (clients 2-5) and the same blocks can be transferred to all these clients. The utilization of CacheCast is highest within this time interval, since the number of overlapping clients is the highest. Hence, when many clients overlap, the performance, compared to FTP, increases.

The end results of the CacheCast optimization is that less bandwidth is consumed in the network and the download time is reduced. The level of gains that can be achieved depends on factors such as the requests rate to the server and the file size (see Chapter 6).

3.7 Reliable transfer

According to the requirements discussed in Section 3.3 a file server should ensure reliable transfer. In Section 2.3 we introduced two techniques for handling reliability, namely *Retransmissions* and *Fountain codes*. The use of retransmissions requires that the file blocks are ordered using sequence numbers, but the content of the blocks remains unmodified. When using Fountain codes, the file blocks are encoded into encoded blocks and reliability is ensured by decoding these encoded blocks. These two techniques of handling reliability integrate well with the transmission procedures explained in Section 3.5. In the following subsections, we discuss how this integration is accomplished. As an introduction to this discussion

we present the choice of transport protocol for the CacheCast file server and mention some implications of this choice.

3.7.1 Transport protocol and implications

In Section 2.2 we introduced the concept of transport protocols. When developing applications for the Internet, there is the choice of which transport protocol to use. The transport protocol which we use for the CacheCast file server is selected based on two criteria: (1) As discussed in Section 2.1, a protocol which preserves message boundaries is required by CacheCast. (2) The mechanisms to enable synchronous transmission handles the data on a block-by-block basis (see Section 3.4). Both of these criteria rule out TCP as a candidate, since this protocol treats the data as a byte stream instead of individual data chunks. UDP and DCCP are message-oriented protocols, i.e., the boundaries between messages are preserved. Therefore, both these protocols are viable alternatives for the CacheCast file server. However, neither of these protocols include reliable transmission, so in either case reliability must be implemented on top of these protocols (on the Application layer). DCCP includes end-to-end congestion control and is therefore preferred over UDP. End-to-end congestion control is necessary to adjust the transmission rate from the server and assure a fair share of network resources (see Section 4.3). In order to use UDP a custom congestion control mechanism would have to be built.

We mentioned in Section 2.3 that the end result of data loss in a network is either a lost packet or a corrupted packet. Corrupted packets are identified by checksums on several layers in the network stack. DCCP includes checksums, and if a wrong checksum is encountered the whole packet is discarded. Therefore, a corrupted packet will never reach the application layer.

Since corrupted packets are handled by DCCP, the reliability mechanism in the CacheCast file server only needs to handle lost packets. There are generally two sources of lost packets. As stated above, a whole packet might be lost due to an error in the network or a packet is discarded because of corruption. On modern wired networks the probability of errors is very low. So this first source of packet loss happens rarely on modern hardware.

The second source of packet loss is a result of queue management on network nodes. When the amount of traffic in the network increases, the queues on the routers fill up. When a queue is full, new arriving packets are dropped, since the queue can not hold them. This scenario is the most common source of packet loss in a network. Often this is referred to as packet drop rather then packet loss.

To summarize, the reliability mechanism in the CacheCast file server resides on the application layer. DCCP is used as transport protocol, which protects against corrupted packets, but does not ensure ordered and reliable data delivery. Therefore, since packet corruption is handled by DCCP (or protocols on lower layers in the network stack) only packet loss needs to be handled by the reliability mechanism.

3.7.2 Transmission of original blocks

The standard procedure for handling packet loss in networks is to retransmit the lost data. TCP adapts this procedure to ensure reliable transfer. In order to use retransmissions there are two conditions which need to be meet: (1) It must be possible to identify data loss and (2) the specific data block which is lost must be identified. As a consequence, the retransmission procedure requires uniquely identifiable blocks and the blocks must be ordered in a known order using sequence numbers (cf. Section 2.3). The procedure of transmitting original blocks explained in Section 3.5 fulfills these conditions. Each block has a unique sequence number and the original block order is known (the transmission order can however be changed).

As explained in Section 2.3 sequence numbers and timers are generally used to identify packet loss. When using sequence numbers the client must deduce, from the sequence numbers of the previously received blocks, which blocks that have been lost. Then the client can notify the server, which will issue a retransmission of the lost blocks. However, as previously discussed in Section 3.4, the client does not know the transmission order of the blocks. Therefore the client can not conclude which blocks that are missing based on the sequence numbers. Therefore, in the CacheCast file server, packet loss are identified using timers. The server uses the following procedure to identify and recover from packet loss:

- The client acknowledges every block it receives by sending an acknowledgement packet to the server. The acknowledgement contains the sequence number for the acknowledged block.
- The server stores the transmission timestamp for each block.
- If the server has not received an acknowledgement for a certain block within a predefined time frame, this block is retransmitted.

In order to support this procedure, some information, here called metadata, must be stored. The server keeps a record of which blocks each client has received. So, for each client, the server stores two values for each block; a *status* and a *timestamp*. The timestamp is set when the block is transferred from the server. In Table 3.2 we describe the different block statuses. When a new client starts a download, the status for all blocks are set to MISSING. Thus, at this time the client can receive every block. If a block has the state RECEIVED, the server knows for sure that this block has been successfully received by the client. However, if a block has the state SENT the server only knows that the block has been sent and that an acknowledgement has not been received. In this case there are three possible scenarios: (1) The block is traversing the network toward the client. (2) The acknowledgement is traversing the network toward the server. (3) Either the block or the acknowledgement has been lost. Only in this third scenario should a retransmission occur. To differentiate between these scenarios, a timer is set for each block. If this timer expires, the block is assumed to be lost. The timer value can be set statically or modified dynamically. In this thesis a timer value of two average Round Trip Times (RTT) is used.

Table 3.2: Block statuses

Status	Description
MISSING	The block has not been sent toward the client. It is certain that the client is missing this block.
SENT	The block has been sent toward the client, but the server has not yet received an acknowledgement for this block.
RECEIVED	The client has acknowledged a successful reception of this block.

The RTT is defined as the time interval from a block is transferred until an acknowledgement for this block is received. The stored timestamp is used to calculate this RTT for each block. An average RTT (aRTT) is stored for each client, which is the average of all RTT values calculated for this client. This aRTT is the one which is used to identify packet loss. If a transferred block has not be acknowledged within $2 \times aRTT$, the block is assumed to be lost and can be retransmitted.

A state diagram of the block statuses and the transitions between the different statuses is depicted in Figure 3.12. A block has initially the status MISSING. The transition to the SENT state is done when the block has been sent from the server. And the block reaches the final state RECEIVED when an acknowledgement for this block is received by the server. The transition named *timeout* happens when the block is in the SENT status and the timer expires. Then the block's status is set to MISSING since we assume that the block is lost. The timeout transition is the mechanism which selects a block for retransmission. In Algorithm 3.1 the procedure of transmitting original blocks is depicted. On line number four the set of clients which are missing *block* is established. All clients where the status of *block* is MISSING is added to the set. There is no distinction whether the block is sent for the first time or it is retransmitted.

In Figure 3.13 the flow of packets when using transmission of original blocks is visualized. When the server receives a request for a file from a client it starts transferring blocks toward the client. The status of the transmitted blocks is set to SENT. The client acknowledges each block it receives. When the server receives the acknowledgments, the metadata for the client is updated, i.e. block status is set to RECEIVED. When the client has received all blocks it notifies that server that the download is complete.

A factor to consider when using the retransmission approach is the amount of storage space needed for metadata. For each connected client metadata must be stored for each file block. When there a many clients and the number of file blocks is large the metadata would consume a significant amount of storage space. However, instead of storing the metadata in the main memory, it could be stored for instance on disc or in a database, to eliminate this issue.

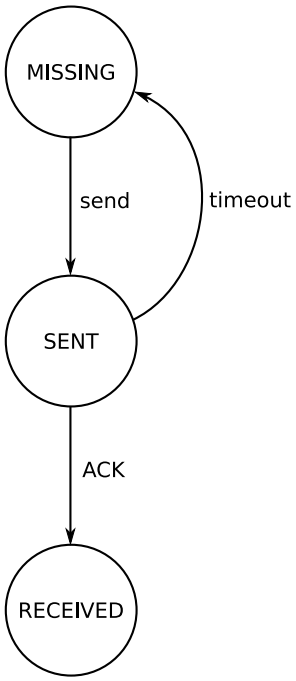

Figure 3.12: State diagram of block statuses

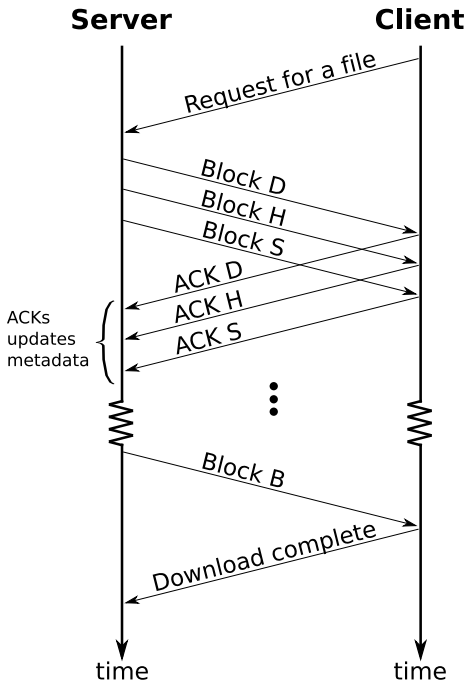

Figure 3.13: Packet flow between client and server when using transmission of original blocks

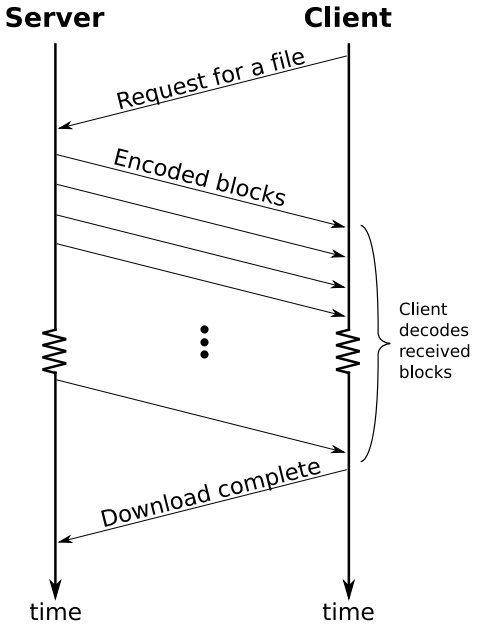

Figure 3.14: Packet flow between client and server when using Fountain codes

3.7.3 Transmission of encoded blocks

The design of the reliability mechanism when using the Fountain codes approach is fundamentally different from the retransmissions approach. As explained in Section 2.3.2, no retransmissions are necessary when using Fountain codes. In Section 3.5, Fountain codes are used to enable synchronous transmission to all clients. A consequence of using Fountain codes is that reliability is handled by the Fountain codes technique. The reliability is achieved by encoding the blocks on the server and transmitting a stream of encoded blocks toward the client. The client collects encoded blocks and decodes the content. Whenever the client has received enough blocks to decode the original file, the file transmission is finished. Hence, the reliability is achieved by collecting enough encoded blocks. In this thesis we assume that the number of encoding blocks a client has to collect is the same as the number of file blocks. In a real-world implementation the number would be a few percent higher.

In Figure 3.14 the packet flow when using Fountain codes is visualized. After the server has received a request from a client, only encoded blocks is transferred from the server to the client. The client can decode the encoded blocks iteratively when they arrive. The fundamental difference compared to the retransmissions approach is the lack of signaling from the client.

3.8 Summary

In this chapter we have discussed the general design considerations for the CacheCast file server. The main purpose of the CacheCast file server is to optimize the file transfer using the CacheCast mechanism. In order to benefit from CacheCast the same data must be transferred to multiple clients synchronously. Sequential transmission order of blocks has proved difficult to combine with synchronous transmission. Thus, the transmission order must be altered to achieve this synchronization. As long as the client is able to reorder the blocks correctly when they are received, this reordering does not influence the reliability of the system.

Two different transmission procedures have been discussed, namely transmission of original blocks and transmission of encoded blocks. The advantage when using encoding is that synchronous transmission can be achieved in scenarios where it is not possible with transmission of original blocks. The Fountain codes approach also simplifies the reliability mechanism, since no retransmissions are necessary. The procedure of transmitting original blocks requires less computation both on the server and client, when compared to encoding.

Chapter 4

CacheCast file server architecture

The basic considerations and design choices made in the previous chapter should be transformed into a system architecture, in order to build a CacheCast file server. In this chapter we present the system architecture of the file server.

A high level overview of a file server with CacheCast support is depicted in Figure 4.1. The core functionality of the file server is to optimize the transmission using CacheCast. As discussed in Section 2.1, the CacheCast mechanism is built for single source multiple destination transfer. When using CacheCast the efficiency gains grows with the number of receivers. The CacheCast file server architecture is therefore designed to support multiple clients downloading a file concurrently.

To support CacheCast, the CacheCast file server architecture differs from a standard FTP server architecture. In an FTP server, the commands offered to the user, like selecting which file to download, is handled by the FTP application on the Application layer. However, the actual file transfer is completely delegated to TCP on the Transport layer. Reliability and flow control is provided fully by TCP.

The CacheCast file server application handles both the file selection and

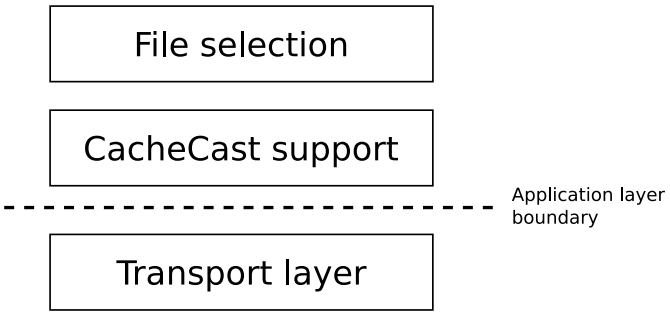

Figure 4.1: Overview of the CacheCast file server

Table 4.1: FTP server vs. CacheCast file server

	FTP server	CacheCast file server
Application Layer	User commands	User commands Rate control Reliability
Transport Layer	Reliability Flow control	Congestion control

a part of the file transfer. Reliability and rate control is provided on the Application layer. The end-to-end congestion control is delegated to DCCP on the Transport layer. In Table 4.1 we summarize the important differences between an FTP server and a CacheCast file server.

4.1 Separation of concerns

A general principle when designing computer applications involve splitting the program into different modules. Each module has a distinct functionality and responsibility. Communication between the modules is done using public interfaces. The various tasks which the application should perform and the relationship among them, is often used as a guideline when deciding upon which modules are necessary.

From the discussion in Section 3 we can derive two major tasks for the CacheCast file server; (1) utilizing CacheCast by enabling synchronous transmission and (2) ensuring reliable transfer. As we have seen, these tasks are closely related. When using original blocks transmission, the blocks must be reordered to achieve synchronous transmission, and reliability is ensured by issuing retransmissions. Thus, both tasks are concerned with selecting blocks for transmission. When using the Fountain codes approach, synchronous transmission and reliability is combined in the same mechanism. The use of Fountain codes to achieve synchronous transmission also ensure reliable transfer. Based on this discussion we choose to create a *Reliability module* which includes the functionality for both synchronous transmission and reliable transfer. The module is a Reliability mechanism optimized for CacheCast.

An important property of data transfers in a network is the transmission rate, i.e. how fast the data is transferred. In Section 3.4 we discussed that hosts in the Internet have different bandwidth capacities. Often servers have higher bandwidth capacities than clients, so it is important that a fast server does not outrun a slower client. In addition, the available bandwidth capacity of a client can vary during the transmission procedure. A network application or protocol should therefore adjust the transmission rate to the available bandwidth capacity of the client. The CacheCast file server includes a module which adjusts the transmission rate, namely the *Rate control module*.

The choice of including a Rate controller demands another module,

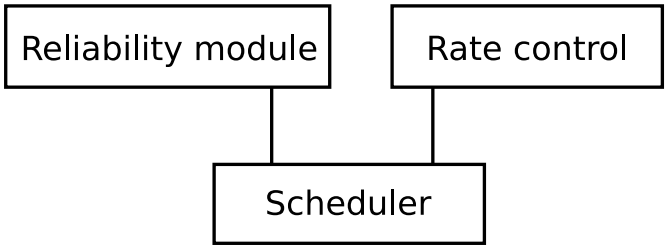

Figure 4.2: Modules in the CacheCast file server

namely a *Scheduling module*. The Scheduler should decide when a block transmission should occur, based on the value of the transmission rate. That is, it should schedule the transmission of the blocks.

Three modules have been identified for the CacheCast file server, namely a *Reliability module*, a *Rate control module*, and a *Scheduler*. The modules and their relationship is visualized in Figure 4.2. The Reliability module selects a block for transmission. The Rate controller adjusts the transmission rate. The Scheduler schedules the transmission of the block selected by the Reliability module, based on the transmission rate from the Rate controller. In the next three sections we explain the details of the functionality of these modules.

4.2 Reliability module

In this thesis we implement and evaluate two transmission procedures, namely transmission of original blocks and transmission of encoded blocks. These procedures are integrated with the two different reliability mechanisms presented in Section 3.7. Therefore, the Reliability module has two implementations, one using retransmissions and one using Fountain codes. As discussed in the previous section, the Reliability module has two tasks, namely achieving synchronous transmission and ensuring reliability. The details of these tasks were discussed in Section 3.5 and Section 3.7. In the Reliability module these two tasks are combined into a single mechanism.

The first implementation of the Reliability module which uses retransmission, is depicted in Figure 4.3. The *block selection* procedure is where a block is selected for transmission. As noted in Section 3.5, in this thesis a block is selected using the Round Robbin algorithm. The file is input to the block selection procedure. The output of the Reliability module is the specific block to transfer and the set of clients to which to transfer this block. The metadata is used to check for possible retransmissions. The functionality of the block selection is depicted in Algorithm 3.1.

The second implementation of the Reliability module uses Fountain codes. This implementation is fundamentally different from the first. In Figure 4.4 an overview of this implementation of the Reliability module is depicted. Again the file in input, but here it is input to the encoder. The

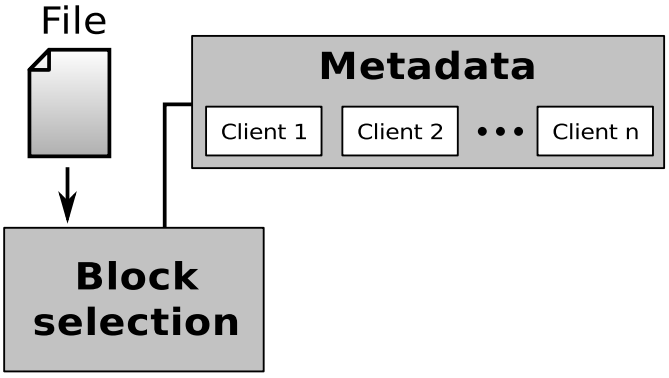

Figure 4.3: Reliability module when using retransmission

encoder is an external procedure which takes a file as input and outputs an encoded block. The specific details of the functionality of the encoder, is beyond the scope of this thesis, but a high level introduction to Fountain codes were given in Section 2.3.2. The output of the reliability module is an encoded block and a set including all clients (cf. Algorithm 3.2). This output is the input into the Scheduler which schedules this transmission of this encoded block. The Scheduler uses the transmission rate to schedule the next transmission. However, the transmission rate can vary during a download procedure. Thus, to account for changes to the transmission rate, a queue is inserted after the encoder. When the encoder is finished encoding a block, it is put into this queue. To supply the Scheduler with encoded blocks as needed, the queue fill rate, i.e. the speed of the encoder, should be higher than the transmission rate.

4.3 Rate control

The CacheCast file server is designed to transfer a file to a group of clients. This group most certainly contains clients with different bandwidth capacities. Therefore, the transmission rate must be adapted to the connected clients. To manage this rate adjustment we build a Rate controller. Its task is to modify the transmission rate based on information about the clients' download speeds.

The Rate controller should modify the transmission rate such that satisfying performance is assured for the whole group of clients. What we mean by "satisfying performance" is that the transmission rate from the server is so high that the clients' bandwidth capacity is fully utilized. The question is what the Rate controller should adapt to, in order to assure satisfying performance? We discuss some possibilities.

The Rate controller could adapt to the average client download speed. Approximately half of the clients would then experience satisfying performance. Another approach could be to adapt to the download speed experienced by the highest number of clients. The problem with these two

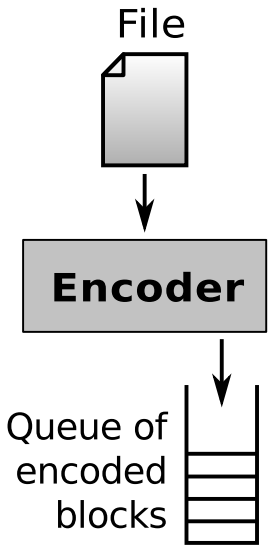

Figure 4.4: Reliability module when using Fountain codes

approaches is that the fastest client(s) will never get satisfying performance (in the second approach the fastest clients would get satisfying performance only if the fastest download speed is experienced by the highest number of clients). We therefore choose another approach. The transmission rate is adjusted according to the fastest client in the group. The rationale for this decision is to make sure that the fastest client receives data at its maximum speed. As for the slower clients, the congestion control algorithm in DCCP will throttle the rate of each client. All clients, regardless of download speed, can therefore be satisfied.

The Rate controller has two major tasks; (1) determining the fastest client and (2) modifying the transmission rate using a rate adaption algorithm. We elaborate on these tasks in the two following sections.

4.3.1 Determining the fastest client

The first task of the Rate controller is to determine the currently fastest client, so the transmission rate can be adjusted to its download speed. In network terminology the term *speed* can be defined in different ways, for instance to denote throughput, bandwidth, latency. In this thesis, we define the term *fastest client* as the client with highest available bandwidth. We use the term *currently fastest client* to emphasize that the role as the fastest client can change due to shifting network conditions. The role as the fastest client must therefore be updated constantly.

To determine the currently fastest client, we use the feedback from DCCP. When a packet is forwarded to DCCP, we get a positive or negative feedback. If the feedback is positive, DCCP has forwarded the packet to the network layer. If the feedback is negative, DCCP has dropped the

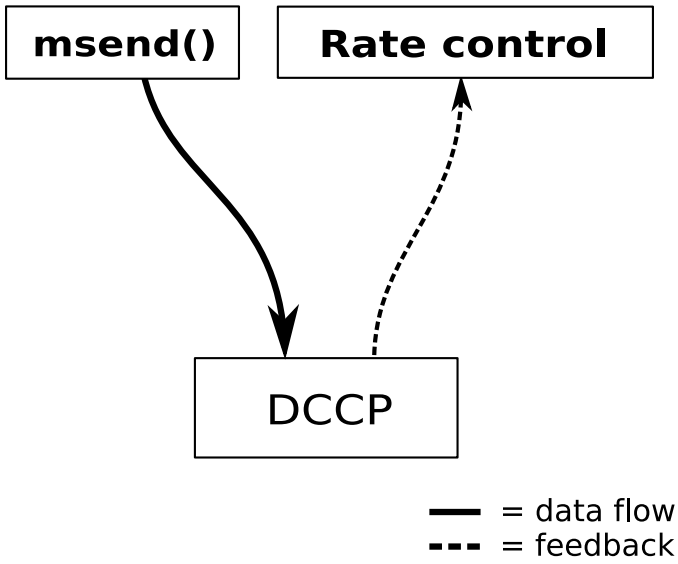

Figure 4.5: Transmission with msend() and feedback from DCCP

Table 4.2: Packet transmission attempt statuses

Status	Description
SENT	The packet was successfully passed to the network layer by DCCP
DROPPED	The packet was dropped by DCCP because of congestion.

packet due to congestion (we do not consider any cases or error). In Figure 4.5 this feedback is visualized as dashed arrows, whereas the solid arrows indicate the data flow. We define the packet forwarding to DCCP as a *packet transmission attempt*. The status of a packet transmission attempt can be either SENT or DROPPED. These statuses and their description are summarized in Table 4.2.

For each client, a record of N such transmission attempt statues, where N is a constant, are stored in the Rate controller. The client with the highest number of SENT statuses is considered as the currently fastest client. We present an example in Figure 4.6. There are three clients with different download speeds, reflected by the number of successful transmission attempts. By counting the SENT statuses, it is clear that Client 2 is the currently fastest client.

The constant N might be adjusted for different scenarios. If N is low, the stored statuses are quickly replaced, making the Rate controller react fast to changes. If N is large, the Rate controller reacts slower to changes. In this thesis we use $N = 100$.

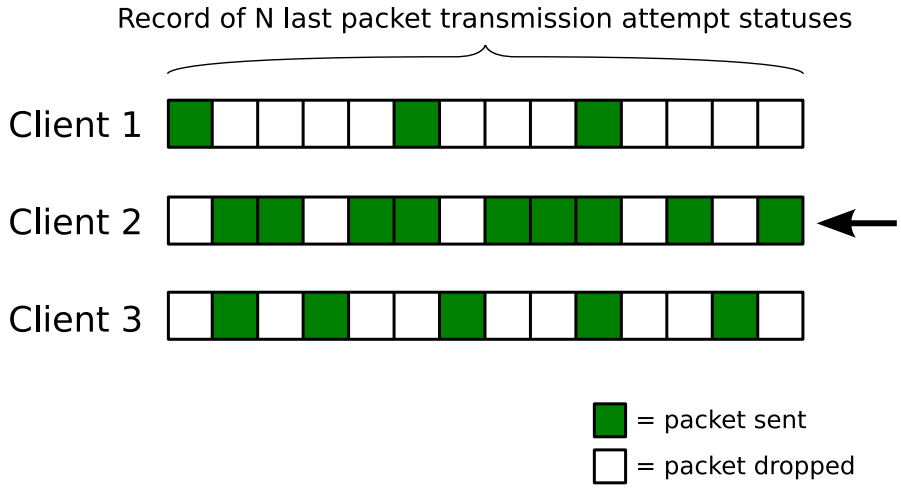

Figure 4.6: Example of determining the fastest client. The arrow indicates the currently fastest client.

4.3.2 Rate control algorithm

The second task of the Rate controller is to perform the actual transmission rate adaptation. In the previous sections we determined which client to consider as the currently fastest one, and in this section we adjust the transmission rate to the speed of this client. Just as we used the feedback from DCCP to measure the fastest client, we use the same feedback to adjust the transmission rate. After a new block transmission attempt, for each client a new transmission attempt status has been added to the record of transmission attempts. The rate adaption procedure is as follows: Investigate the last transmission attempt status of the fastest client. If this status is SENT, increase the transmission rate. If the status is DROPPED, decrease the transmission rate.

The rationale here is to adapt to the congestion control mechanism of DCCP. If a DROPPED status is reported by DCCP the congestion control algorithm has prevented the transmission of a new packet. We therefore assume that the transmission rate is too high and that it needs to be decreased. On the other hand, if a SENT status is reported, we know that the last packet was successfully forwarded by DCCP and the transmission rate is increased. The rationale of increasing the transmission rate is to always assume that more bandwidth is available. To utilize this "assumed bandwidth" the transmission rate must be increased.

This rate adaption procedure is similar to the modification of the congestion window in TCP. When a congestion event is experienced, the congestion window size is decreased. When there is no congestion the size of the congestion window is increased. The end result is a modified transmission rate. Standard TCP congestion avoidance algorithms are Additive Increase Multiplicative Decrease (AIMD) [13] algorithms. AIMD

is a feedback control algorithm which preserves fair share for multiple flows over a contended link.

The rate control algorithm in the CacheCast file server is an AIMD algorithm and it works as follows: If the last packet was sent, add to the transmission rate r an increase factor $I > 0$. If the last packet was dropped, decrease the rate by multiplying it with a decrease factor $D \in (0,1)$ (see Formula 4.1).

$$r = \begin{cases} r + I, & \text{if last packet was sent} \\ r \times D, & \text{if last packet was dropped} \end{cases} \tag{4.1}$$

The transmission rate r is the rate used by the Scheduler to transmit a block to a group of clients and r is adjusted to the fastest client within the group. The slower clients within the group can not receive data at this rate, but, as previously explained, the adaption to the slower clients is handled by DCCP. However, it may seem unnecessary to adapt the transmission rate in the first place, since DCCP can do this adaption for all clients, even the fastest one. A theoretical "infinite rate" could be used and the congestion control in DCCP would throttle this rate for every client. The problem with this approach is that unnecessary resources are used on the server. The processor would be constantly busy trying to transmit data, even though no clients would be able to receive it. When using Fountain codes, the encoder could not possibly keep up with this "infinite rate". Based on these arguments there is a rationale for controlling the transmission rate, and that is limiting the resource consumption on the server. In addition, there is no point in transferring data at a higher rate than the rate of the fastest client.

The transmission rate must be set to an initial value. In this thesis the initial transmission rate is set to 64 kb/s.

4.4 Scheduler

The main task of the Scheduling module is to schedule the transmission of blocks, based on the transmission rate. In addition, it also executes the actual transmission procedure.

The scheduling is done on the basis of the transmission rate. The scheduling procedure is as follows: After the current block transmission has finished, a new block is scheduled for transmission after t seconds, where t is calculated from the transmission rate and the packet size using Formula 4.2. In this formula the transmission rate is in bits per second, so the packet size is multiplied by 8 to get the packet size in bits instead of bytes.

$$t = \frac{\text{packet size} * 8}{\text{transmission rate}} \tag{4.2}$$

In Figure 4.2 the relationship between the three modules of the file server is visualized. The Scheduler is connected to both the reliability mechanism and the Rate control module. The complete procedure of the Scheduling module is as follows:

1. Get a new block and a set of clients from the Reliability module
2. Transmit the block to the clients in the set using the msend() function
3. Only for transmission of original blocks: Update the metadata by changing the status for the block to SENT
4. Tell the Rate controller to update the transmission rate
5. Schedule the next transmission using the updated transmission rate

When transmission of original blocks is used, the output block from the Reliability module is a file block which is transferred to the set of clients missing it. When Fountain codes is used the output from the Reliability module is an encoded block which is transferred to all clients. After the msend() function has been executed the Rate controller has received new feedback based on the last transmission attempt (cf. Figure 4.5). In point four above this new feedback information is used to modify the transmission rate.

4.5 Multiple files

So far in this thesis, the focus has been on multiple clients downloading a single file from the server. However, a file server should also support multiple files being downloaded concurrently. In a standard FTP server, multiple files can be stored and multiple clients can download multiple files concurrently.

In the CacheCast file server the clients downloading the same file is bundled into a *group*. When there are multiple files being downloaded, there is one group of clients per file. As carefully discussed in Section 3.4, the rationale for grouping clients is to achieve synchronous transmission.

The download of a single file to a group of clients is called a *file session*. For each file being downloaded, there is one file session. For each file session there is one group of clients. All the previously discussed mechanisms, such as rate control, reliability mechanism etc., is in regard to a single file download. When there are multiple concurrent file downloads, there is one such mechanism per file session. When the CacheCast file server receives a request for a file, it checks whether this file is currently being downloaded, i.e. a file session exists for the requested file. If not, a new file session is created, and a new group consisting of the new recipient, is added to it. If, on the other hand, there exists a file session for the requested file, the new recipient is appended to the existing group of clients.

In Figure 4.7 and Figure 4.8 we present the detailed architecture for the CacheCast file server, when using original block transmission and Fountain codes, respectably. The figures visualize the full server architecture. In the following chapters, the implementation and evaluation of this architecture is explained. As noted in the previous paragraph, there is one Rate controller per file session, which means that each file session has a separate transmission rate. The transmission rate r_j for each file session j is adjusted

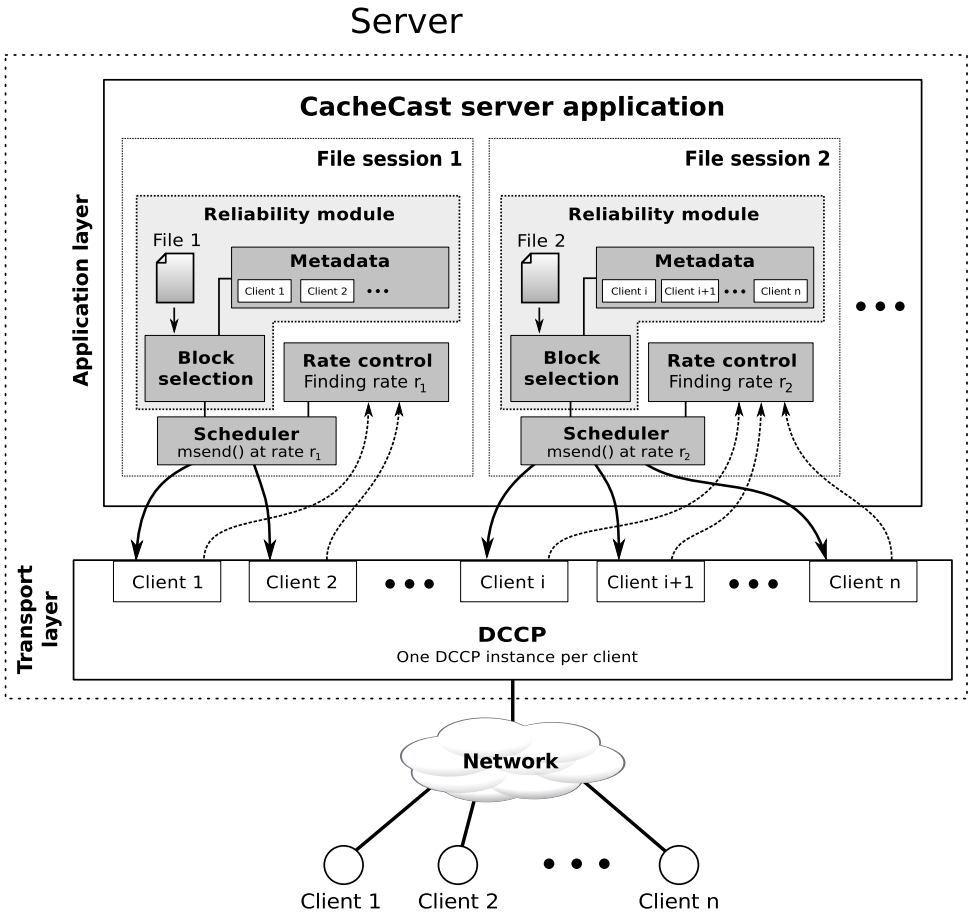

Figure 4.7: CacheCast file server architecture when using original block transmission

to the group of clients downloading file j. As an example of a group, *Client 1* and *Client 2* are downloading *File 1* and therefore constitutes *Group 1*. There is also one Scheduling module per file session, which schedules transmissions based on the rate r_j. Each client is connected to the server with a separate unicast DCCP connection. Even though the clients are batched into groups of clients, the data is sent on separate DCCP connections.

4.6 Client support

The design of the CacheCast file server presented above demands a specific client design. In this thesis the focus has been on the server design, but in this section we briefly explain the necessary components in the client application.

Basically, the client design depends on which transmission procedure is used. In this thesis we explore two such procedures, namely transmission of original blocks and transmission of encoded blocks. Therefore, two implementations of the client is necessary. In both implementations the client instantiates the file transfer by sending a request to the server.

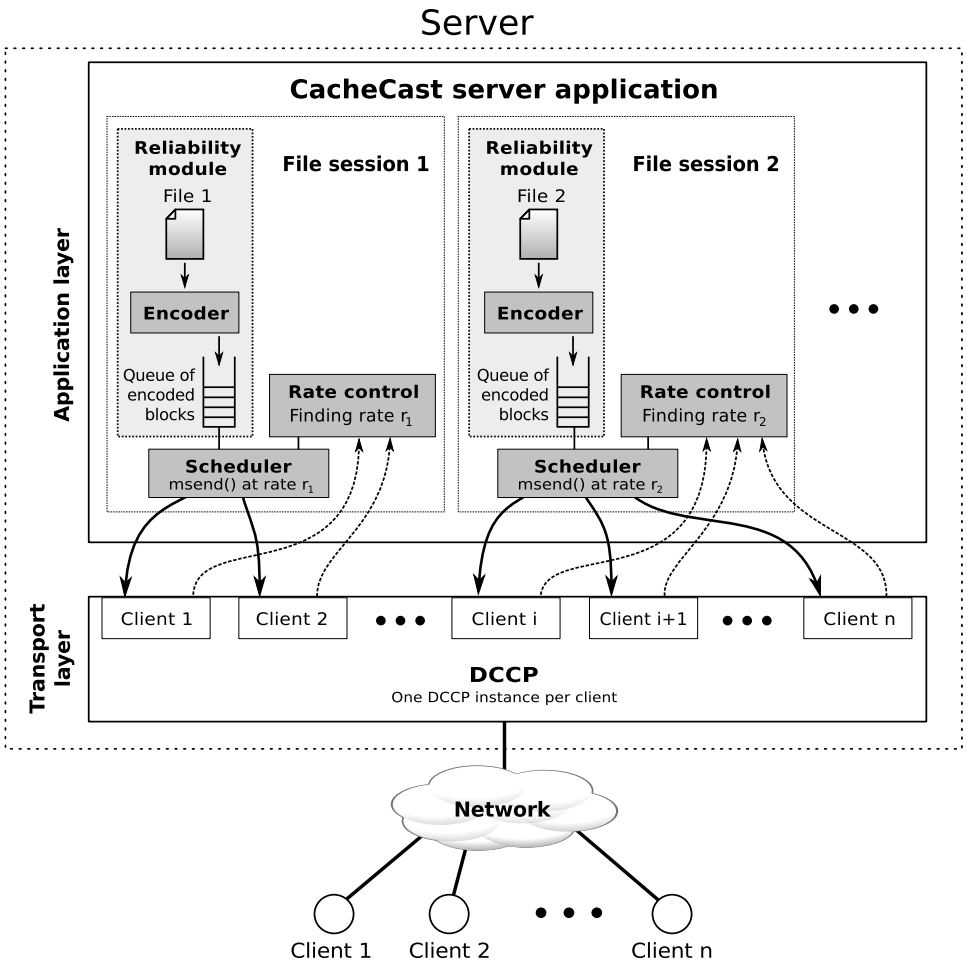

Figure 4.8: CacheCast file server architecture when using Fountain codes

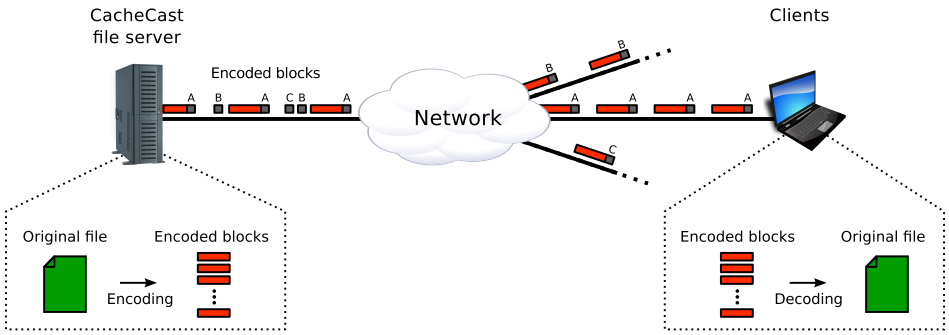

Figure 4.9: High level overview of the CacheCast file server, network and client

4.6.1 Transmission of original blocks

When transmission of original blocks is used the client has generally two tasks; reordering the received blocks and transmitting acknowledgments to the server for each successfully received block. Each block contains a unique sequence number and these sequence numbers are used to reorder the blocks correctly. An acknowledgment is transferred back to the server whenever a new block is received, and it contains the sequence number for the received block. When the client has correctly received all blocks, it notifies the server that the download procedure has been successful.

4.6.2 Transmission of encoded blocks

When the Fountain codes approach is used, the tasks of the client is different than when transmission of original blocks is used. In this approach, the client receives encoded blocks from the server. These encoded blocks must be decoded into the original file. When using Fountain codes the client has only one major task; decoding the received encoded blocks. This decoding can be done iteratively when new encoded blocks are received. Hence, there will not be any significant overhead caused by the decoding process (cf. Section 2.3.2). When the client has received enough encoded packets to decode the original file, it notifies the server that the download procedure is finished.

In Figure 4.9 we present an overview of the whole system of CacheCast file server, network, and client. The Fountain codes approach is depicted in this figure. On the server the original file is encoded into encoded blocks, which are transferred using the CacheCast mechanism toward the clients. On the clients the encoded blocks are collected and decoded into the original file.

4.7 Summary

In this chapter the general design considerations discussed in chapter 4 were used to build a system architecture for the CacheCast file server. The

"Separation of concerns" principle was used to split the functionality of the CacheCast file server into three independent modules, namely *Reliability module*, *Rate control* and *Scheduler*. The Reliability module selects a block for transmission and constructs the set of clients missing this block. The Rate controller adapts the transmission rate to the fastest connected client. The Scheduling module schedules the transmission of the block from the Reliability modules to the set of clients, based on the transmission rate from the Rate controller. All these modules cooperate in fulfilling the requirements for a CacheCast supported file server.

In Section 4.5 the issue of multiple concurrent file downloads was discussed. The concept of a client *group* was described and to separate the download of multiple files, the concept of a *file session* was introduced. In the last section we briefly discussed the architecture of the client-side of the system in order to obtain a complete description of a file transfer system from server to client.

Chapter 5

Implementation

In this chapter we present the implementation of the CacheCast file server, which is based on the design and architecture discussed in the previous chapter. The system is implemented in order to evaluate its performance. There are different methods for evaluating network systems. Common examples include *mathematical analysis*, *simulation*, *emulation* and *real-world implementation*. When doing mathematical analysis, the system is represented by mathematical models. Calculation and analysis is performed on these models. This method is frequently used as a first attempt of showing the benefits of the system. The use of simulations is a way of evaluating the system in a context similar to the real-world. A simulator contains models for real-world phenomenons and protocols, and is a framework in which the system is implemented and evaluated. The implementation of a system in a simulator often needs to be simplified compared to a real-word implementation. In order to create a realistic simulation environment care must be taken to ensure that the main functionality of the system is well represented in the simulation.

The evaluation method which gives the most realistic results is doing measurements in a real-world implementation of the system. However, such an implementation usually requires a large amount of resources and are time consuming. Therefore, emulation is often used to simplify parts of the evaluation. In emulation, simulation and real-world implementation is combined. Parts of the system are implemented completely, while other parts are simulated.

In this thesis we have chosen to evaluate the CacheCast file server using simulation. The whole system is built in the ns-3 network simulator framework [6]. Ns-3 aims to simulate the functionality of a real network. Therefore, the implementation of the CacheCast file server in ns-3 also aims to model a real-world implementation of the file server.

Implementing applications or protocols in simulators has positive and negative consequences. These consequences must be taken into account in order for the simulations to be reliable. In ns-3, no computation on the nodes is modeled. Hence, only the effects on the network and data flow are available for study through our implementation of the CacheCast file server. The computational complexity of the implementation does not impact the

simulation results. In a real-world implementation, the algorithms would be optimized to limit the computational load on the server and client.

The implementation of a system in a simulator is often simpler and more high-level than a real-world implementation. In order to build realistic simulations it is important to take into account the abstractions and simplifications made by both the simulation framework and the system implementation. The implementation presented in this chapter has been built as close to the design as possible, without taking shortcuts and simplifying the functionality. Thus, the implementation of the CacheCast file server in ns-3 is close to how a real-world implementation would be built.

In real networks, user applications on end-hosts are the main source of traffic between the computers in the network. In ns-3, computers are represented by nodes and applications installed on the nodes create packets and transmit these packets through the network. Thus, as in a real network, applications in ns-3 are the source of traffic in the simulated network. In our implementation no actual data files are transferred. The file transfers are simulated by transferring packets with random data. This does not influence the performance of the system, since the content of the transferred files is irrelevant.

The ns-3 network simulator includes support for the major protocols used in the Internet, such as IP, TCP, and UDP. It lacks, however, native support for DCCP. DCCP is the transport protocol used in the CacheCast file server, so in order to build a realistic evaluation environment, DCCP support is important. A project called *Network Simulation Cradle (NSC)* [4] enables real-world operating system's network stacks to be used within simulators. NSC includes support for the Linux kernel version 2.6.29, which includes DCCP support. A project called *NS3-NSC-DCCP* [1] supports DCCP through NSC in ns-3. This support is used in our implementation of the CacheCast file server. However, the CacheCast mechanism requires that network packets are sent from the server without delay [43]. Therefore, the DCCP protocol has to be slightly modified to disable delaying of packets. Delayed packets are simply dropped. This modification is added to the Linux DCCP implementation. In this thesis we use version 3.10 of the ns-3 network simulator.

In the following section, we describe the implementation of the CacheCast mechanism in ns-3, while the rest of this chapter contains the implementation details for the server and client-side of the system.

5.1 CacheCast implementation in ns-3

In order to build simulations to evaluate the CacheCast file server, the CacheCast mechanism has been implemented in ns-3. The implementation of CacheCast in ns-3 follows closely the implementations by Srebrny in Linux and the Click modular router [43]. This implementation (as in the general design of CacheCast) consists of three main components; *server support*, *Cache Management Unit (CMU)* and *Cache Store Unit (CSU)*. The placement of these components in a ns-3 simulation is depicted in Figure

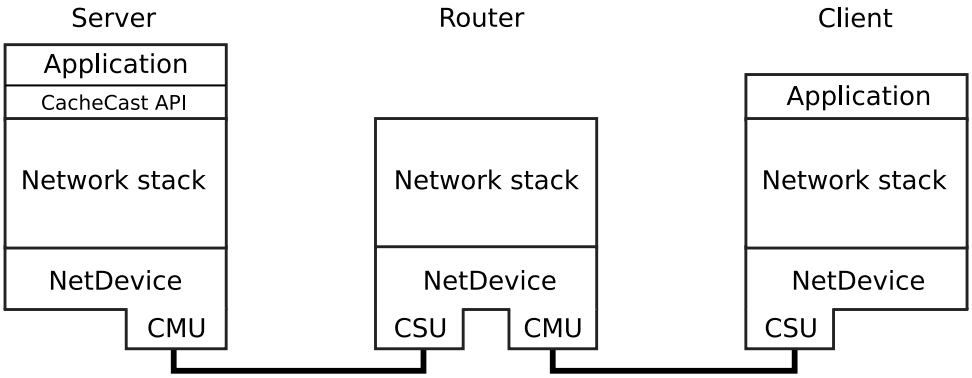

Figure 5.1: CacheCast components in ns-3

5.1. The CMU and CSU are included within a NetDevice on the channel entry and channel exit, respectably. The server support is realised as a layer between the application and the network stack. The details of the functionality of the components are explained in Section 2.1. In the following we explain the specific implementation details of the components in ns-3.

5.1.1 Server support

In the server support implementation of CacheCast in Linux, there are two components; a msend() system call offered to applications and an underlying packet modification mechanism. The main task of the packet modification mechanism is to ensure that the packets are transferred in a tight packet train. This is handled by storing the packets temporarily and transmitting all packets onto the channel in a batch. This mechanism is necessary because of the multiprogramming nature of modern operating systems. However, in ns-3 there is no operating system. Processing on the nodes is not modelled in the simulation and all tasks are executed in sequence without interruption. Therefore, a specific mechanism to ensure a tight packet train is not necessary. The batching of sockets in the application domain is sufficient to form tight packet trains.

The Msend() function is offered by an API in a module called *CacheCast*, which applications can import and use. The C++ class name in ns-3 for this module is also CacheCast. In Figure 5.1, this module is visualized as *CacheCast API*. The Msend() function has two main tasks; transmit a packet to a set of receivers and identify the packets as cacheable packets. The code for Msend() is available in Listing 5.1. The receivers are represented by a vector of sockets. A pointer to a packet called packet and the vector of sockets called sockets, are the parameters to Msend(). For each socket a copy is made of packet, a CacheCastTag is added to this packetCopy, and packetCopy is sent on the socket. The CacheCastTag identifies packetCopy as a cacheable packet, and stores the payload ID and the payload size. It is used by the CMU to identify and truncate the CacheCast packets correctly (see Section 5.1.2). The function GetNewPayloadId() is global for the node and

calculates a new unique payload ID. The packet is sent using Socket::Send(). If the packet could not be sent, the index of the current socket is added to a vector failedIndex. This vector can be examined by the application to identify which sockets failed to send. The functions BeginFailedIndex() and EndFailedIndex() returns a C++ iterator to the front and back of the failedIndex vector, respectably. Msend() returns true if the packet was successfully sent on all sockets.

Listing 5.1: Msend() function

```
1  bool CacheCast::Msend(Ptr<Packet> packet,
2      vector<Ptr<Socket> > sockets) {
3    bool success = true;
4    uint32_t socketIndex = 0;
5
6    for(Iterator socket = sockets.begin();
7        socket != sockets.end(); socket++) {
8      Ptr<Packet> packetCopy = packet->Copy();
9
10     /* Add CacheCast tag */
11     CacheCastTag tag(GetNewPayloadId(), packetCopy->GetSize());
12     packetCopy->AddPacketTag(tag);
13
14     if((*socket)->Send(packetCopy) < 0) {
15       failedIndex.push_back(socketIndex);
16       success = false;
17     }
18     socketIndex++;
19   }
20   return success;
21 }
```

5.1.2 Cache Management Unit (CMU)

The main task of the CMU is to remove redundant packet payloads and add the CacheCast header. The CMU is installed in the NetDevice on the channel entry (cf. Figure 5.1) so it captures the CacheCast packets before they are transferred onto the channel. The CMU does not store the actual packet payload, but manages this storage which is located in the CSU. It contains a table of which packet payloads are stored in the CSU. This table is managed by two maps, namely tableIdToIndex and tableIndexToItem. The indexes of the CSU are stored in the first map, while the second map contains values necessary to manage the cache correctly (e.g. a timestamp). These values are stored in a struct TableItem in the map. The code for the packet handling mechanism of the CMU is available in Listing 5.2. First in CacheManagementUnit::HandlePacket() a check is performed for whether the packet will fit in the cache. After calculating a universally unique id for the packet, a search for this id in the table is executed. If the id is present (cache hit), the payload is removed from the packet and the CacheCast header is modified. The timestamp for this id is also checked, in order to invalidate this entry if it is more than 1 second old (this is done according to the CacheCast design to allow payload IDs to wrap around). Next, if the id was not found in the table (cache miss), enough room is created for the payload in the table,

and the new packet info is added. The CacheCast header is modified with the correct table index and added to the packet. As in the general design of CacheCast, the CacheCast header in ns-3 contains the payload ID, payload size and index in the cache.

Listing 5.2: HandlePacket() function in CMU

```
1  bool CacheManagementUnit::HandlePacket(Ptr<Packet> p) {
2    bool timeout = false;
3    CacheCastTag tag;
4    p->PeekPacketTag(tag);
5    CacheCastHeader cch(tag.GetPayloadId(), tag.GetPayloadSize(), 0);
6
7    /* Check if there are enough slots */
8    uint32_t slotsCount = (tag.GetPayloadSize() != 0) ?
9      (tag.GetPayloadSize() - 1) / slotSize + 1 : 1;
10   NS_ASSERT_MSG(slotsCount <= size,
11       "CacheCast packet is too large for the CSU");
12
13   /* Get IPv4 address of packet */
14   Ipv4Header ipHdr;
15   uint32_t ipRead = p->PeekHeader(ipHdr);
16   uint32_t addr = ipHdr.GetSource().Get();
17
18   /* Calculate universally unique id */
19   uint64_t id = ((uint64_t) addr << 32) | tag.GetPayloadId();
20
21   /* Search for id in table */
22   Iterator it = tableIdToIndex.find(id);
23
24   /* Cache hit */
25   if(it != tableIdToIndex.end()) {
26     /* Check if element it too old */
27     TableItem &item = tableIndexToItem[(*it).second];
28     if(Simulator::Now().GetSeconds() - item.timeStamp > 1.0) {
29       tableIdToIndex.erase(item.id);
30       timeout = true;
31     }
32     else {
33       p->RemoveAtEnd(tag.GetPayloadSize());
34       cch.SetPayloadSize(0);
35       cch.SetIndex((*it).second);
36     }
37   }
38   /* Cache miss */
39   if(it == tableIdToIndex.end() || timeout) {
40     uint32_t index = currIndex;
41     cch.SetIndex(index);
42
43     for(uint32_t i = 0; i < slotsCount; i++) {
44       TableItem &item = tableIndexToItem[currIndex];
45
46       if(item.idInSlot) {
47         tableIndexToItem[currIndex].idInSlot = false;
48         tableIdToIndex.erase(item.id);
49       }
50
51       /* Cache replacement is a round robbin scheme */
52       currIndex = (currIndex + 1) % size;
```

```
53        }
54        tableIndexToItem[index].id = id;
55        tableIndexToItem[index].idInSlot = true;
56        tableIndexToItem[index].timeStamp =
57          Simulator::Now().GetSeconds();
58        tableIdToIndex[id] = index;
59      }
60    p->AddHeader(cch);
61    return true;
62  }
```

5.1.3 Cache Store Unit (CSU)

The CSU is installed in the NetDevice at the channel exit, as visualized in Figure 5.1. Between the CMU and CSU the CacheCast packets form the packet train structure. While the CMU handles the management of the caching mechanism, the CSU is where the packet payloads are actually stored. The CSU has two tasks: (1) Whenever a CacheCast packet containing a payload is seen, store the payload of this packets in the cache. (2) Whenever a truncated packet is seen, append the stored payload to this packet. In addition, the CacheCast header is removed from the packet.

In Listing 5.3 the code for the HandlePacket() function of the CSU is available. The payload size and a unique id are stored in a map cache, while the payloads are stored in a map store. Initially the CacheCast header is removed from the packet. After some preliminary checks of the packet size and cache index, a universally unique id is calculated (the same id as in the CMU). Then there is a check for whether a truncated packet arrived (a packet containing only headers). If this is the case, the correct payload, given by the index in the CacheCast header, is retrieved from the cache and appended to the packet. Due to limitations in ns-3, a temporary packet tmpPacket must be created and this packet is appended to the received headers. An integrity check is performed on the stored and calculated id. If they are not the same, the packet should be discarded, so the function returns false. If, on the other hand, the received packet is a complete packet with a payload, the payload should be stored in the cache. Therefore, the cache is updated to hold this new payload. Correct id and payload size is stored in cache and the actual payload contents are copied from the packet onto the store.

Listing 5.3: HandlePacket() function in CSU

```
1  bool CacheStoreUnit::HandlePacket(Ptr<Packet> p) {
2    CacheCastHeader cch;
3    p->RemoveHeader(cch);
4
5    /* Check if there are enough slots */
6    uint32_t slotsCount = (cch.GetPayloadSize() != 0) ?
7      (cch.GetPayloadSize() - 1) / slotSize + 1 : 1;
8    NS_ASSERT_MSG(slotsCount <= size,
9        "CacheCast packet is too large for the CSU");
10   NS_ASSERT_MSG(cch.GetIndex() < size,
11       "CacheCast index is too large");
12
13   /* Get IP address of packet */
```

```
14    Ipv4Header ipHdr;
15    uint32_t ipRead = p->PeekHeader(ipHdr);
16    uint32_t addr = ipHdr.GetSource().Get();
17
18    /* Calculate universally unique id */
19    uint64_t id = ((uint64_t) addr << 32) | cch.GetPayloadId();
20
21    /* Only header arrived */
22    if(cch.GetPayloadSize() == 0) {
23      TableItem &item = cache[cch.GetIndex()];
24      if(item.id != id) {
25        return false;
26      }
27      StoreItem storeItem = store[cch.GetIndex()];
28      Ptr<Packet> tmpPacket = Create<Packet> (storeItem.contents +
29        storeItem.offset, item.payloadSize);
30      p->AddAtEnd(tmpPacket);
31    }
32    /* Full CacheCast packet arrived */
33    else {
34      cache[cch.GetIndex()].id = id;
35      cache[cch.GetIndex()].payloadSize = cch.GetPayloadSize();
36
37      /* Store packet contents */
38      Iterator it = store.find (cch.GetIndex());
39      if (it != store.end()) {
40        delete[] (*it).second.contents;
41      }
42      uint8_t *tmp = new uint8_t[p->GetSize()];
43      p->CopyData(tmp, p->GetSize());
44      StoreItem storeItem (tmp, p->GetSize() - cch.GetPayloadSize());
45      store[cch.GetIndex()] = storeItem;
46    }
47    return true;
48  }
```

The code for both the CMU and the CSU in Listing 5.2 and Listing 5.3, follows closely the code from Srebrny [43] written for the Click modular router. The design of the code is generally the same but the syntax and some logic has been adapted to the ns-3 framework.

5.2 Server-side implementation overview

In this section, we give an overview of the CacheCast file server implementation in ns-3. The important classes and data structures are introduced, in order to prepare for the description of the implementation of the different modules.

The CacheCast file server is implemented as a ns-3 application. The class CacheCastFileServer encapsulates the whole CacheCast file server and is derived from the native ns-3 Application class. The Application class contains functions and attributes required by ns-3 applications, such as functions to start and stop the application. The CacheCastFileServer is mainly a container for the file sessions, as visualized in Figure 4.7 and Figure 4.8. Its main task is to handle new requests from clients. When the CacheCastFileServer receives

a new request from a client, this client is added to the correct group based on which file it is requesting. The actual file transfer functionality reside in each file session.

5.2.1 FileSession

The class FileSession represents a file session, i.e., the download procedure of a single file to multiple clients. The CacheCastFileServer contains a list of all the current FileSessions, implemented as a vector vector<Ptr<FileSession> > filesessions. Each FileSession handles the transmission of one file to a group of clients. A client is added to a FileSession by the CacheCastFileServer when it arrives to the server. An object Client is created for each connected client and this object is stored in a vector vector<Ptr<Client> > clients in the FileSession class. This vector contains all clients currently downloading the file.

In each FileSession the size of the file and the block size is stored in the variables fileSize and blockSize, respectably. These variables are used to calculate sequence numbers and specify the size of the transmitted blocks. In this thesis we use a block size of 1024 bytes (1 KB).

The functionality of the CacheCast file server is split into three modules, as discussed in Section 4. These modules reside within each file session. Therefore, in this implementation most of the modules' functionality lies in the FileSession class. In Listing 5.4 the contents of this class is shown. The contents of the three modules are highlighted using comments. The specific implementation details of these modules are discussed in Section 5.3. The public interface of the FileSession class is used by the CacheCastFileServer to create new file sessions and add clients. A client is removed from the file session using RemoveClient() when the download procedure for this client is finished.

Listing 5.4: FileSession class

```
1  class FileSession : public Object {
2    public:
3      FileSession(uint32 blockSize, uint32 fileSize);
4      void AddClient(Ptr<Client> c);
5      void RemoveClient(Ptr<Client> c);
6      uint32 GetFileSize();
7
8    private:
9      Ptr<BlockSelection> blockSelection;        // Reliability module
10     vector<Ptr<Client> >                       //
11       GetClientsMissingBlock(uint32_t block); //
12
13     double transmissionRate;                   // Rate control
14     void UpdateTransmissionRate();             //
15
16     CacheCast cachecast;                       // Scheduler
17     void ScheduleNext();                       //
18     void SendPacket();                         //
19
20     vector<Ptr<Client> > clients;
21     uint32_t blockSize;
22     uint32_t fileSize; // in blocks
23 };
```

5.2.2 Client

The Client class contains the server-side data for a client, such as socket and metadata. In addition, the record of transmission attempt statuses for the Rate controller is stored in this class. Whenever a new client connects to the server to download a file, a new Client object is created. There is one such object for each client request. As explained in the previous section, the Client objects are stored in the clients vector in class FileSession. In Listing 5.5 the contents of the Client class is listed. The FileSession in which the Client object reside, is stored in the variable fileSession. This information is available for the Client object to remove itself from the file session when the download procedure is finished. As discussed in Section 3.7, the Reliability module uses a client's round trip time (RTT) to calculate if a retransmission should occur. Therefore, the Client class calculates the RTT for each block and stores the average RTT in the variable averageRTT. This calculation happens in the HandleRead() function which is explained below. The transport layer socket for the client connection is stored in the variable socket.

Listing 5.5: Client class

```
1  class Client : public Object {
2    public:
3      Client(Ptr<Socket> socket);
4      void SetFileSession(Ptr<FileSession> fileSession);
5      Ptr<Socket> GetSocket();
6      void UpdateMetadata(uint32_t block, BlockStatus status);
7      bool IsMissingBlock(uint32_t block);
8
9      /* Rate controller specific */
10     int32_t GetNumSuccessfulTransmissions();
11     void AddTransmissionAttemptStatus(bool status);
12     TransmissionStatus GetLastTransmissionAttemptStatus();
13     void ResetLastTransmissionAttemptStatus();
14     enum TransmissionStatus {NOT_SENT, DROPPED, SENT};
15
16   private:
17     void HandleRead(Ptr<Socket> socket);
18     Ptr<FileSession> fileSession;
19     Ptr<Socket> socket;
20     Metadata metadata;
21
22     /* Calcuted RTT for this client */
23     int32_t averageRTT;
24     uint32_t calcRttTotal;
25     uint32_t calcRttSamples;
26
27     /* Rate controller specific */
28     TransmissionStatus lastTransmissionAttemptStatus;
29     list<bool> transmissionAttemptStatuses;
30     int32_t numSuccessfulTransmissions;
31 };
```

Metadata

The metadata, i.e. block status and timestamp for each block, is stored in a class Metadata. In this class a vector vector<BlockInfo> blocks contains an element for each file block. The BlockInfo class stores the block status, which is either MISSING, SENT, or RECEIVED, and a timestamp for when the block was transferred from the server. In the Client class the metadata is stored in the variable metadata. The function void UpdateMetadata() updates the status of block and sets the timestamp to the current simulation time.

HandleRead()

The HandleRead() function is where the acknowledgments from the client-side of the data connection is received. It is treated as a callback function in the ns-3 framework and is registered in the socket using socket->SetRecvCallback (MakeCallback (&Client::HandleRead, this)). Whenever the socket receives new data, the HandleRead() function is called. The code for this function is available in Listing 5.6. The function enters a while loop where it checks and retrieves any new packets which are available on the socket. Each acknowledgement sent from the client contains a header AckHeader. This header contains a single attribute, namely block, which is the sequence number of the acknowledged block. The HandleRead() function checks to make sure that the received packet is in fact an acknowledgement. Next, there is a simulation specific test which checks whether Fountain codes are used in the current simulation. From Figure 3.14 it is clear that, when using Fountain codes, the client only signals back to the server when it has successfully received the whole file. An acknowledgement, when using Fountain codes, is the indication that the entire file has been successfully received by the client. Therefore, if FOUNTAIN_CODES is true, the function only removes the client from the file session and returns. The file transmission for this client is finished.

If FOUNTAIN_CODES is false, transmission of original blocks is used in the simulation. Therefore, the metadata is updated with the information in the acknowledgement, i.e. the status of the acknowledged block is set to RECEIVED. Then, the RTT value for the acknowledged block is calculated and the average RTT is updated. In the end, the HandleRead() function checks if the download procedure is finished, i.e. all blocks have been acknowledged. The function Metadata::GetNumAcknowledgedBlocks() returns the number of acknowledged blocks. It takes into account that in rare cases, the same block can be acknowledged several times. So multiple acknowledgments for the same block is only counted once. If the number of acknowledged blocks is equal to the number of blocks in the file, the whole file has been successfully received by the client. This client is then removed from the file session.

Listing 5.6: Client::HandleRead() function

```
1 void Client::HandleRead(Ptr<Socket> socket) {
2   Ptr<Packet> packet;
3
```

```
4     while(packet = socket->Recv()) {
5       uint8_t buf[packet->GetSize()];
6       packet->CopyData(buf, packet->GetSize());
7
8       if(AckHeader::IsAckHeader(buf)) {
9         if(FOUNTAIN_CODES) {
10          fileSession->RemoveClient(this);
11          return;
12        }
13        AckHeader ackHdr(buf);
14        UpdateMetadata(ackHdr.block, RECEIVED);
15
16        /* Update the average RTT */
17        int64_t now = Simulator::Now().GetMilliSeconds();
18        int64_t then = metadata.GetTimeStamp(ackHdr.block);
19        int64_t blockRTT = now - then;
20        calcRttTotal += blockRTT;
21        averageRTT = calcRttTotal / ++calcRttSamples;
22      }
23
24      /* All blocks have been acknowledged, i.e. download finished */
25      if(metadata.GetNumAcknowledgedBlocks() ==
26          fileSession.GetFileSize()) {
27        fileSession->RemoveClient(this);
28        return;
29      }
30    }
31  }
```

In Figure 5.2 we present an UML class diagram of the classes explained above and their relationship. Only the most important attributes and functions are available in this diagram. It is worth to notice the multiplicity on the associations. The CacheCastFileServer may contain zero file sessions. This happens when no clients are downloading any files. However, the FileSession must contain at least one Client. When the first client requests a file from the server a new file session is created. If all clients finished downloading, the file session is no longer needed and it is removed.

The functions and attributes of the Client class which have not yet been mentioned, is explained in the description of each module below.

5.3 Modules

The major functionality of the CacheCast file server reside in three modules; *Reliability module*, *Rate control*, and *Scheduler*. In this section we elaborate on the implementation of these modules in the ns-3 framework. All these modules are implemented in the FileSession class.

5.3.1 Reliability module

The Reliability module ensures that a file is transferred correctly to the client. In the implementation of the CacheCast file server in ns-3, an actual data file is not being transferred. The file server only simulates a file transmission by sending blocks with random data.

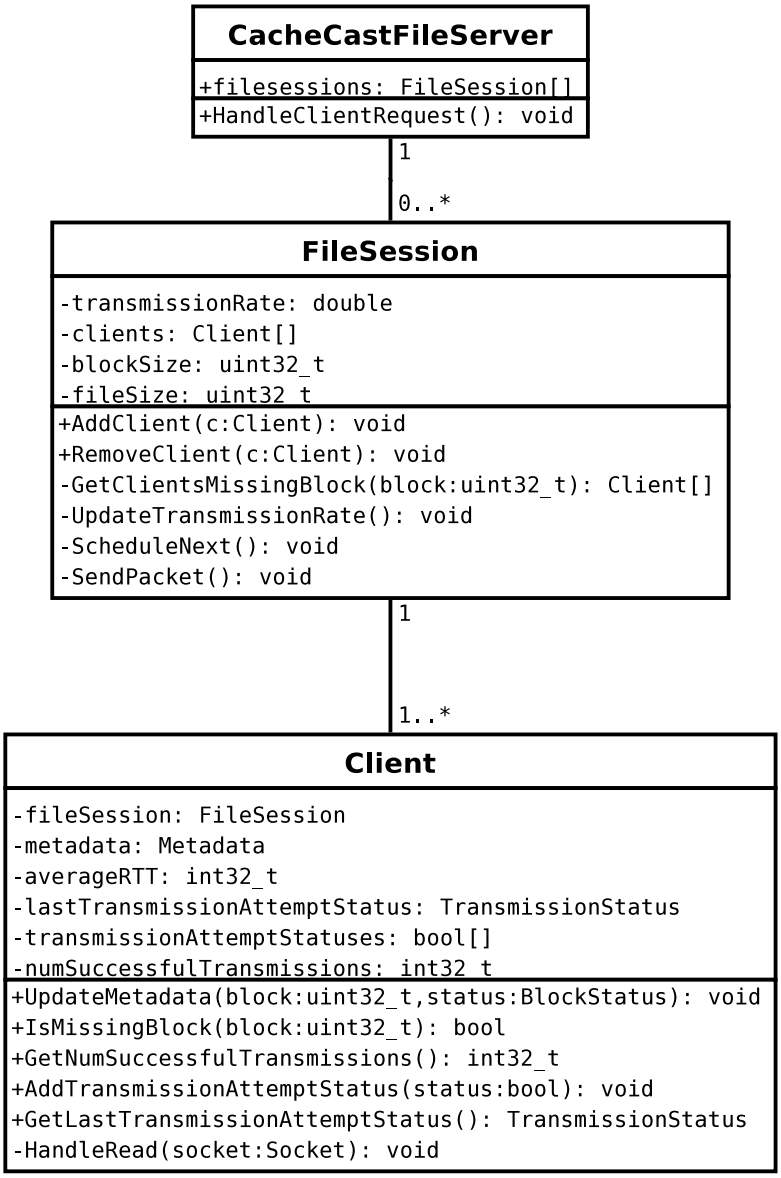

Figure 5.2: UML class diagram of the main server-side classes

If transmission of original blocks is used, each block is labeled with a unique sequence number in order for the client to calculate when all blocks have been received. If transmission of encoded blocks is used, the blocks are assumed to contain encoded data, and no sequence numbers are used. No encoding/decoding is actually performed, since no data file is being transferred.

Transmission of original blocks

The implementation of the Reliability module consists of two elements, blockSelection and a function vector<Ptr<Client> > GetClientsMissingBlock(uint32_t block), as seen in Listing 5.4. The class BlockSelection controls which blocks to transmit. As mentioned in Section 3.6, the block selection algorithm used in this thesis is Round Robbin. BlockSelection contains a function uint32_t GetNextBlock() which returns block indexes in a Round Robbin fashion.

The function GetClientsMissingBlock() takes the block index as an argument and returns the set of clients which are missing this block. The function Client::IsMissingBlock(uint32_t block) checks whether the client is missing a certain block. It is used in GetClientsMissingBlock() to check which clients should be added to the set. The code for this function is available in Listing 5.7. Firstly, the block status is retrieved from the metadata. Then, assuming that block has been sent before, we compute the time past since this block was transferred. The timePast variable is used to check whether this block's timer has expired. If the timer has in fact expired, the status of this block is set to MISSING (cf. Figure 3.12). At the end, the function returns true whenever the block status is MISSING. The status is MISSING either because the block has not been transferred before or because the timer has expired.

Listing 5.7: Client::IsMissingBlock() function

```
1  bool Client::IsMissingBlock(uint32_t block) {
2    BlockStatus status = metadata.GetStatus(block);
3    int64_t now = Simulator::Now().GetMilliSeconds();
4    int64_t then = metadata.GetTimeStamp(block);
5    int64_t timePast = now - then;
6
7    /* Check for expired timer */
8    if(status == SENT && timePast > 2 * rtt) {
9      status = MISSING;
10     metadata.setStatus(block, status);
11   }
12   return status == MISSING;
13 }
```

Transmission of encoded blocks

If the Fountain codes approach is used, no block selection is necessary since it is assumed that encoded blocks are transferred, and these are created by an encoder (in a real-world implementation). Then, only the number of blocks received by the client matters. In addition, an encoded block is transferred

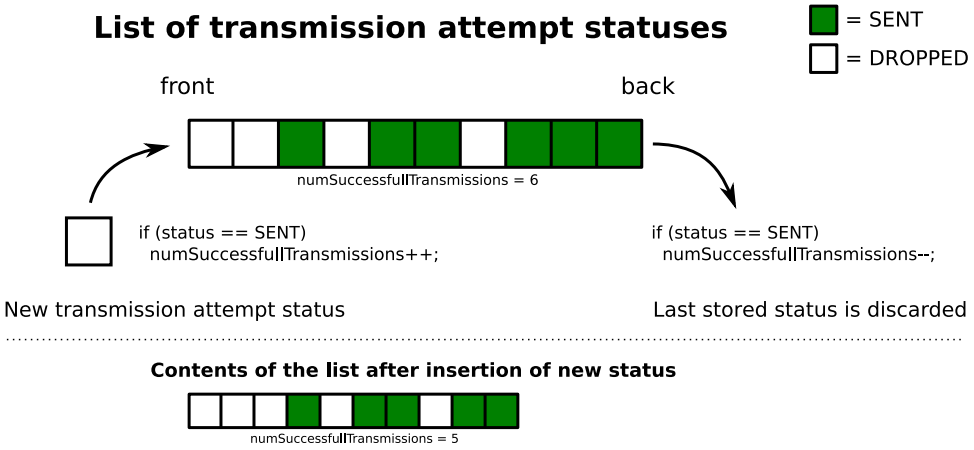

Figure 5.3: Algorithm for updating numSuccessfulTransmissions

to all clients. Therefore, GetClientsMissingBlock() is not used with Fountain codes. The functionality of the Reliability module when using Fountain codes is then to transmit a block with random data to all connected clients.

5.3.2 Rate control

The task of the Rate controller is to modify the transmission rate based on the bandwidth capacity of the fastest client, as explained in Section 4.3. The Rate controller is implemented in the function FileSession::UpdateTransmissionRate(). The fastest client is determined according to the description in Section 4.3. The list of transmission attempt statuses is represented by list<bool> transmissionAttemptStatuses in the Client class. A boolean value of true represents a SENT status and a false value represents a DROPPED status.

In order to find the fastest client, the number of successful transmission attempts in the list is counted. The result is stored in the variable numSuccessfulTransmissions. However, the list does not need to be traversed to find the number of SENT statuses, as long as this value is updated whenever a new status is added to the list. The algorithm which updates numSuccessfulTransmissions is depicted in Figure 5.3. When a new transmission attempt status is added to the list, the last status is discarded. This way the size of the list remains constant (except in the startup phase since the list is initially empty). If the added transmission attempt status is SENT, numSuccessfulTransmissions is incremented. If the last status, which is discarded, is SENT, numSuccessfulTransmissions is decremented. Using this algorithm, numSuccessfulTransmissions mirrors the correct number of SENT statuses in the list. In the example in the figure, the new status is DROPPED so numSuccessfulTransmissions is not increased. The discarded status is SENT so numSuccessfulTransmissions is decreased. The end result is that numSuccessfulTransmissions has been decreased from six to five after the new status has been added.

A new transmission attempt status is added to transmissionAttemptStatuses using the function Client::AddTransmissionAttemptStatus(). The task of updating the list is performed by the Scheduler after a transmission procedure (a call to Msend()) is finished (see Section 5.3.3). The function Client::GetLastTransmissionStatus() returns the last status, which is used to decide whether to increase or decrease the transmission rate. The Client::GetNumSuccessfulTransmissions() function returns numSuccessfulTransmissions.

In Listing 5.8, the code for UpdateTransmissionRate() is available. Firstly, the fastest client is determined by comparing the number of successful transmissions for each client. Then, the transmission rate is either increased or decreased based on the value of the last transmission attempt status, as explained in Section 4.3. In this thesis INCREASE_FACTOR is set to 16 (kb/s) and DECREASE_FACTOR is set to 0.8.

Listing 5.8: FileSession::UpdateTransmissionRate() function

```
1  void FileSession::UpdateTransmissionRate() {
2    Ptr<Client> fastestClient = clients.front();
3
4    /* Find the fastest client */
5    for(Iterator c = clients.begin() + 1; c != clients.end(); c++) {
6      if((*c)->GetNumSuccessfulTransmissions() >
7          fastestClient->GetNumSuccessfulTransmissions()) {
8        fastestClient = *c;
9      }
10   }
11
12   TransmissionStatus status =
13     fastestClient->GetLastTransmissionAttemptStatus();
14
15   /* Update transmission rate */
16   if(status == DROPPED)
17     transmissionRate += INCREASE_FACTOR;
18   else if(status == SENT)
19     transmissionRate *= DECREASE_FACTOR;
20 }
```

5.3.3 Scheduler

The Scheduling module is realised through two functions in the FileSession class, namely ScheduleNext() and SendPacket(). The task of the Scheduler is to transmit the block from the Reliability module to all clients missing it, with the rate specified by the Rate controller. Thus, the Scheduler connects the three modules as depicted in Figure 4.2. The SendPacket() function handles most of the functionality. The ScheduleNext() function schedules a new block transmission at some time in the future, calculated based on the transmission rate. The implementation of the SendPacket() function is available in Listing 5.9. The block and set of clients from the Reliability module is retrieved in the beginning of the function. The variable block is the block selected for transmission by the Reliability module, while the variable clientSet is the set of clients missing this block. A block is encapsulated in a ns-3 packet called packet before transmission.

Listing 5.9: SendPacket() function

```
1  void FileSession::SendPacket() {
2    vector<Ptr<Client> > clientSet;
3    Ptr<Packet> packet;
4    uint32_t block;
5
6    if(FOUNTAIN_CODES) {
7      clientSet = clients; // the set of all clients
8      packet = Create<Packet>(blockSize);
9    }
10   else { /* Transmission of original blocks */
11     block = blockSelection->GetNextBlock();
12     clientSet = GetClientsMissingBlock(block);
13
14     /* Create packet with sequence number */
15     DataHeader hdr(block);
16     uint8_t data[blockSize];
17     hdr.Serialize(data);
18     packet = Create<Packet>(data, blockSize);
19   }
20
21   /* Create vector of sockets */
22   vector<Ptr<Socket> > socketSet;
23   for(uint32_t i = 0; i < clientSet.size(); i++) {
24     socketSet.push_back(clientSet[i]->GetSocket());
25   }
26
27   cachecast.Msend(packet, socketSet);
28
29   /* Update metadata and transmission attempt status */
30   Iterator failedIndex = cachecast.BeginFailedIndex();
31   for(uint32_t i = 0; i < clientSet.size(); i++) {
32     if(failedIndex != cachecast.EndFailedIndex() &&
33         i == *failedIndex) {
34       clientSet[i]->AddTransmissionAttemptStatus(false);
35     } else {
36       clientSet[i]->UpdateMetadata(block, SENT);
37       clientSet[i]->AddTransmissionAttemptStatus(true);
38     }
39   }
40
41   UpdateTransmissionRate();
42
43   /* Reset the clients last transmission attempt status */
44   for(uint32_t i = 0; i < clientSet.size(); i++)
45     clientSet[i]->ResetLastTransmissionAttemptStatus();
46
47   ScheduleNext();
48 }
```

If Fountain codes are used, clientSet is the set of all clients, and packet is filled with random data. No block is retrieved from the Reliability module. The static ns-3 function call Create<Packet>(blockSize) creates a ns-3 packet with size blockSize. If transmission of original blocks is used, the Reliability module is used to select a block and a set of clients, through the functions GetNextBlock() and GetClientsMissingBlock(). As the previously mentioned AckHeader, the structure DataHeader also contains a block attribute which

is the sequence number of the transferred packet. The block variable in SendPacket() is set as the sequence number in DataHeader. The function call Create<Packet>(data, blockSize) creates a packet with the contents of the data array (which contains the DataHeader), and the size blockSize.

The Msend() function call takes two parameters; the packet to send and a set of sockets on which to transfer copies of this packet. Therefore, a set of sockets called socketSet is created from the set of clients. The Msend() function is called with the arguments packet and socketSet. After Msend() has returned we investigate the return values from DCCP. As noted in Section 5.1.1, these return values are stored in the CacheCast object, and they can be investigated through an iterator and the functions BeginFailedIndex() and EndFailedIndex(). The return value from DCCP is a boolean value where a value of false represents that the packet was dropped by DCCP and true represents a successfully forwarding. Therefore, the variable failedIndex contains the index for the sockets in socketSet where the transmission attempt was unsuccessful, i.e. the packet was dropped by DCCP. For each client in the set clientSet the status of the transmission is checked. If the transmission was unsuccessful a DROPPED transmission attempt status is added to the client's transmissionAttemptStatuses with the function call clientSet[i]->AddTransmissionAttemptStatus(false). If, on the other hand, the packet was successfully forwarded by DCCP, a SENT transmission attempt status is added to the client's list and the metadata for the transferred block is updated, i.e. the block status is set to SENT.

With the updated transmission attempt statuses the Rate controller can modify the transmission rate. Thus, the Scheduler calls UpdateTransmissionRate(). In order to prepare for the next block transmission the last transmission attempt status is reset for each client in clientSet. That is, lastTransmissionAttemptStatus is set to NOT_SENT. At the end of the function the ScheduleNext() function is called. This function has two related steps, (1) calculate the next block transmission time t, based on the transmission rate, and schedule the SendPacket() function. Formula 4.2 is used for the calculation of the new transmission time. The actual scheduling is performed using the built-in ns-3 scheduler. The function call Simulator::Schedule(t, &FileSession::SendPacket, this) schedules the SendPacket() function after t seconds.

This discussion of the ScheduleNext() function concludes the description of the server-side implementation of the system. In the next section we briefly discuss the client-side implementation.

5.4 Client-side implementation

The design of the CacheCast file server demands a specialized design on the client-side, as discussed in Section 4.6. Since our implementation of the system is in a network simulator, the task of the client is straightforward. The functionality of the client depends on which transmission procedure is used, either transmission of original blocks or transmission of encoded blocks (Fountain codes). If transmission of original blocks is used, the server transmits blocks containing sequence numbers. The task of the client is then

to keep track of which sequence numbers it has received. When the client has received all sequence numbers, the download procedure is finished. The client acknowledges each block it receives.

If transmission of encoded blocks is used, the functionality of the client becomes even simpler. In a real-world implementation, the client would have to collect a certain amount of encoded blocks in order to decode the original file. In our implementation the task of the client is only to count the number of received blocks. We assume that the total number of original blocks in the file (fileSize) is enough to decode the whole file (cf. Section 2.3.2). The client notifies the server when enough blocks have been received.

The CacheCast file client is, as the file server, implemented as a ns-3 application named CacheCastFileClient. This application is installed on the client nodes in the simulations. In Listing 5.10 the code for the HandleRead() function of CacheCastFileClient is available. The function receives the blocks from the server. It has a similar structure as the HandleRead() function on the server-side. The packet is retrieved from the socket and there is a check for whether the packet contains a DataHeader. If Fountain codes are used numEncodedBlocks is incremented and this variable is compared with the number of file blocks (fileSize). If enough blocks have been received the CacheCastFileClient notifies the server that the download procedure is complete, by sending an acknowledgement back to the server. The acknowledgments are sent using the SendACK() function, which adds a sequence number to the acknowledgement (if present) and transmits this acknowledgement packet toward the server. The CacheCastFileClient then terminates by scheduling a call for the function StopApplication().

If transmission of original blocks is used, the DataHeader is extracted from the packet. The CacheCastFileClient contains a set set<int32_t> blocks which contains all received sequence numbers. The sequence number of the newly received block is compared against the items in the set, and it is added to the set if not present. This check ensures that multiple receptions of the same block is only counted once. The received block is then acknowledged using SendACK(dataHdr.block). At the end of the function there is a check for whether all the blocks in the file have been received. If this is true, the download procedure is finished and the CacheCastFileClient terminates.

Listing 5.10: HandleRead() function in CacheCastFileClient

```
1  void CacheCastFileClient::HandleRead(Ptr<Socket> socket) {
2    Ptr<Packet> packet;
3
4    while(packet = socket->Recv()){
5      uint8_t buf[packet->GetSize()];
6      packet->CopyData(buf, packet->GetSize());
7
8      if(DataHeader::IsDataHeader(buf)) {
9        if(FOUNTAIN_CODES) {
10         numEncodedBlocks++;
11         /* Check if enough encoded packets have been received */
12         if(numEncodedBlocks == fileSize) {
13           /* ACK = download completed */
14           SendACK();
15           Simulator::ScheduleNow(
```

```
16              &CacheCastFileClient::StopApplication, this);
17            return;
18          }
19        } else { /* Transmission of original blocks */
20          DataHeader dataHdr(buf);
21          if(blocks.find(dataHdr.block) == blocks.end()) {
22              blocks.insert(dataHdr.block);
23          }
24          /* Send ACK for the received block */
25          SendACK(dataHdr.block);
26
27          /* Check if the whole file has been received */
28          if(blocks.size() == fileSize) {
29              Simulator::ScheduleNow(
30                &CacheCastFileClient::StopApplication, this);
31              return;
32          }
33        }
34      }
35    }
36  }
```

5.5 Summary

In this chapter, the implementation of the CacheCast file server has been introduced and explained. The CacheCast file server and the CacheCast mechanism are fully implemented in ns-3 to closely simulate a real-world system. The CacheCast file server implementation follows the architecture described in Chapter 4. The ns-3 network simulator models packets flowing through a network, but it does not model computation on the nodes. Therefore, our implementation covers the network effects of the CacheCast file server, but not the computational load on the server and clients. The implementation of the client-side of the system has been briefly discussed to obtain a description of the full implementation of the system in ns-3.

Included with this thesis is a DVD containing the source code for the CacheCast file server. In the root directory of the DVD is a file README.txt which includes some information about the source code.

Chapter 6

Evaluation

The CacheCast file server has been implemented in ns-3 with the purpose of evaluating the network performance of the system. When developing a new network protocol or application, an evaluation is necessary to validate the functionality and the system design, and to examine the benefits of the new system. In this chapter we present the evaluation of the CacheCast file server and discuss and analyse the experiment results.

In the first section we give a general introduction to the evaluation and experiment setup. In Section 6.2 through Section 6.5, we present four evaluations, each testing a different part of the file server. Section 6.6 provides some further insights of the system. In Section 6.7, we discuss the issue of not being able to evaluate the computational complexity of the system. In the final section we summarize the chapter.

6.1 Introduction

As previously mentioned, in this thesis we have chosen to evaluate the system using simulation. There are several reasons for why we have implemented the CacheCast file server in the ns-3 network simulator. We discuss the most important ones.

The CacheCast technique has previously been implemented in Linux and in the Click modular router by Srebrny [43], so a real-world implementation of CacheCast exists. However, the evaluation of the CacheCast file server requires experiments with many clients in order to examine the performance of the system. This thesis does not include the resources to build a large real-world testbed with dozens of computers. Therefore, due to the project's time and resource constrains, a real-world implementation of the CacheCast file server has not been built. In the ns-3 simulator it requires minimal work to set up a simulation with arbitrary number of clients. The network topology and network parameters are easily changed in order to modify the experiments. This makes ns-3 a practical framework in which to evaluate the CacheCast file server in different scenarios.

The ns-3 simulation framework contains a rich API, which enable us to fully implement both the CacheCast mechanism and the CacheCast file server in ns-3. This is important in order to create a realistic testbed, without

having to simplify the system design.

As opposed to a real-world implementation, where it often isn't straightforward to collect measurements, ns-3 are designed to easily measure just about any parameter in a simulation. This enables us to quickly study how the system behaves under various conditions. Basically, ns-3 provides for us a simple to use, yet powerful, framework, in which to evaluate various aspects of the CacheCast file server easily.

Nevertheless, the choice of evaluating the CacheCast file server in a simulator has some implications on what is possible to evaluate. The major difference between an implementation in a simulator and a real-world implementation is that the simulator does not model the computational load on the network hosts. Therefore, in our evaluation we only study the network effects from using the CacheCast file server. The computational complexity of the implementation will not affect the results of the simulation. This drawback of not being able to model the computational load is discussed in Section 6.7.

6.1.1 Evaluation parts

The evaluation of the CacheCast file server in this thesis is a network performance evaluation. We study how the system performs in a network compared to FTP and observe its effect on the network resources. The evaluation is an isolated evaluation, i.e. the impact from other network traffic is not examined. This choice has been made to study the effects of the CacheCast file server on the network exclusively. The impact from CacheCast on other network traffic and how other traffic affects CacheCast has been throughly studied by Srebrny [43].

The evaluation of the CacheCast file server contains four parts. In each part we evaluate one aspect of the file server. Here we briefly introduce these evaluations, while the specific experiment details and the results and analysis are presented in Section 6.2 through Section 6.5.

Rate control evaluation The goal of the rate controller is to utilize as much as possible of the available bandwidth to the currently fastest client. Therefore, in the first evaluation we examine whether the rate controller modifies the transmission rate according to the rate control algorithm (cf. Section 4.3.2) and that it adapts to the currently fastest client. The evaluation is performed both using transmission of original blocks and transmission of encoded blocks.

Download time evaluation From a user perspective the important factor of a file download is generally the amount of time it takes to finish. A user would like the download to finish as fast as possible. Therefore, the second evaluation is concerned with the download time experienced by the user. The download time when using the CacheCast file server is compared to the download time when using FTP, in order to study the differences of the two systems from a user perspective.

Bandwidth consumption evaluation The performance gains from CacheCast is a result of removing redundancy from network links. In the third

evaluation we therefore investigate the impacts of the CacheCast file server on the bandwidth consumption. As in the previous evaluation the results are compared to FTP.

Fairness evaluation When there are multiple files on a server, clients can download different files concurrently. The network resources should be equally shared, both among the clients downloading the same file and across groups of clients downloading different files. In the forth and final evaluation, the fairness among concurrent file downloads is studied.

6.1.2 Common experiment setup

The experiments discussed in the following sections all include some common setup. In this section we explain the choice of network topology, discuss ns-3 specific simulations setup and introduce the workload of the system.

Topology

In order to perform experiments on a simulated network the choice of network topology must be made. There are various kinds of well known basic topologies, such as *bus*, *line*, *star*, *ring*, *tree* and *mesh*. A network topology called *dumbbell* [25] is a combination of the *line* and *tree* topologies. This topology contains a set of nodes on the left connected to a router and a set of nodes on the right connected to a router. The right and left routers are connected through a single link. All traffic between the right and left nodes are transferred over this link. The link is often called the *bottleneck link*, since it is shared by all clients, and contention between the clients' data flow occurs on this link. The dumbbell topology is widely used to study the effect of a link shared by many hosts, especially when evaluating TCP congestion control mechanisms. In this thesis we use a simplified version of the *dumbbell topology* depicted in Figure 6.1. A single server on the left is connected to a router. This router connects a number of clients on the right to the server. This topology is used to model the single source multiple destination scenario.

In a more realistic network topology the paths connecting the server and router consists of multiple links connected through intermediate nodes. However, as stated by Hespanha et al., "... to analyze congestion control mechanisms, one often ignores the existence of all the intermediate links, except for the *bottleneck link* ..." [25]. The rationale here is that all clients share a single link and this is where most congestion occur and where all clients have to compete for the available bandwidth. Thus, adding intermediate nodes to the dumbbell topology would not affect the data flow in the network significantly.

The main task of this thesis is to study the performance of the system when using CacheCast. CacheCast is a link layer technology and in the simplified dumbbell topology all traffic from the server to the clients has to traverse the first hop link. Therefore, the topology in Figure 6.1 is useful

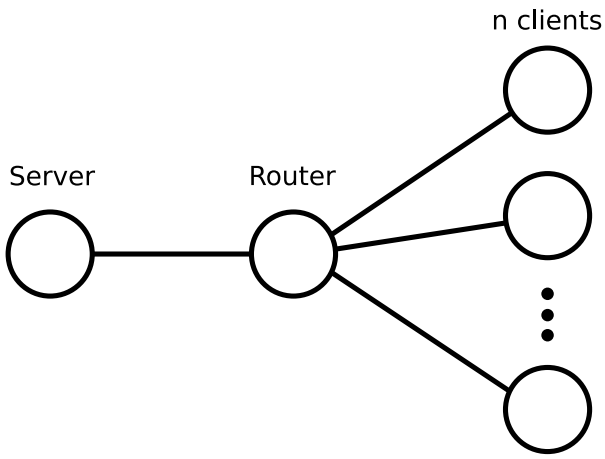

Figure 6.1: Topology used in the evaluations

Table 6.1: Client downlink speed distribution

Downlink speed (kb/s)	64	256	768	1500	3000	5000
Share (%)	2.8	4.3	14.3	23.3	18.0	37.3

to study the effects of the CacheCast mechanism in the network. In addition, when evaluating the fairness between groups of clients, a shared link is necessary in order for the clients to compete for available bandwidth. The selected topology contains such a shared link.

The same dumbbell topology is used in all four evaluations. The number of clients is changed based on the type of evaluation. In order to model the diversity of downlink speeds of hosts in the Internet, the bandwidth of the links between the router and the clients is chosen based on measurement results from Huang et al. [28], who measured the distribution of client downlink speeds in a media server. In this thesis we use this distribution as a reference point for the number of clients with a certain downlink speed. The specific downlink speeds and their share is listed in Table 6.1. As an example, 37.3 % of the clients have the highest downlink speed, namely 5 Mb/s, while 2.8 % have a speed of 64 kb/s. All clients with the same downlink speed constitute a *downlink speed group*.

The first hop link (between server and router) has a bandwidth capacity of 10 Mb/s. This choice is based on the client downlink speed distribution, so that congestion occurs on this link. Even though the link does not have the smallest bandwidth capacity, we still call it the *bottleneck link* in the simulated network.

The end-to-end propagation delays in the simulated network, from the server to the clients, is distributed uniformly between 30 ms and 50 ms. The network parameters here discussed are used in order to simulate a medium sized ISP network.

ns-3 simulation setup

All the experiments in this thesis are conducted in the ns-3 network simulator. Simulations in ns-3 are created as *simulation scripts* and they are, as the core ns-3 framework, written in C++. To simplify the creation of simulation scripts, *helper classes* are available, which handle the details of the simulation setup. Simulations in ns-3 generally consists of the following steps:

1. Create all the nodes in the simulation.
2. Connect the nodes by building a simulated topology.
3. Set the network parameters.
4. Install network stacks on the nodes, and applications on the end-hosts.
5. Start the simulation.

The CacheCast file server simulation follows the same structure. The code for the first step above is available in Listing 6.1. A single server and router node is created using the CreateObject<Node>() function. The numClients specify the number of clients in the simulation and numClients nodes are created and stored in NodeContainer clients.

Listing 6.1: Create nodes

```
1 Ptr<Node> server = CreateObject<Node>();
2 Ptr<Node> router = CreateObject<Node>();
3 NodeContainer clients;
4 clients.Create(numClients);
```

The server and router are connected and the network parameters are set with the code in Listing 6.2. The CacheCastHelper makes it easy to install CacheCast support on a channel. The bandwidth on the channel is set using SetDeviceAttribute() while the propagation delay is set using SetChannelAttribute(). The first hop channel has a delay of 20 ms, while the other channels' delay is uniformly distributed between 10 ms and 30 ms, which combined gives the total end-to-end propagation delay. The CacheCastHelper::Install() function creates a channel with the specified parameters and connects the server and router to this channel. In addition, it installs a Cache Management Unit on the channel entry and a Cache Store Unit on the channel exit.

Listing 6.2: Connect nodes and set network parameters

```
1 CacheCastHelper serverHelper;
2 serverHelper.SetDeviceAttribute("DataRate", StringValue("10Mb/s"));
3 serverHelper.SetChannelAttribute("Delay", StringValue("20ms"));
4 serverHelper.Install(server, router);
```

As for the other nodes, they are connected to the router in same way as the server, but instead of using a CacheCastHelper a PointToPointHelper is used. It contains the same public interface as the CacheCastHelper. CacheCast

support is only installed on the channel from the server to the router. This channel represents the core network, while the channels from the router to the clients represent the "last mile" links. Based on the assumption that packet loss happens very rarely on wired links, no explicit packet loss is introduced on the links. The only packet loss in the simulation is related to dropped packets, due to congestion. There is no background traffic in the simulated network (see Section 6.1.1).

In the fourth step above a network stack is installed on each node. The code for this procedure is available in Listing 6.3. The InternetStackHelper is used to simplify the installation process of stacks. DCCP is installed on the server and clients only. The Linux implementation of DCCP is selected using InternetStackHelper::SetL4Protocol().

Listing 6.3: Install network stacks

```
1 InternetStackHelper stack;
2 stack.Install(router);
3 stack.SetL4Protocol("ns3::NscDccpL4Protocol",
4     "Library", StringValue("liblinux-2.6.29.so"));
5 stack.Install(server);
6 stack.Install(clients);
```

In addition to installing network stacks, the fourth step includes the procedure of installing applications on the end-hosts. In our simulations a CacheCastFileServer application is installed on the server node and a CacheCastFileClient application is installed on each client node. The applications are created using Create<CacheCastFileServer>() and Create<CacheCastFileClient>(). The applications are installed on the nodes with the Node::AddApplication() function.

When the simulation setup is complete the simulator is started using Simulator::Run(). A network simulator is deterministic and gives the same results for equal parameters each time it is run. The simulations in this thesis are therefore run multiple times with different seeds to the random number generator, in order to achieve independent trials. The seed is set using SeedManager::SetRun(uint32_t run). The results of the simulation trials are averaged into a final result. The standard deviation is given when appropriate.

Workload

In order to study the behavior of the simulated system, a representative traffic pattern of a possible real-world system should be chosen. In the simulations in this thesis all clients connect to the server at a point in time and request a file. In this scenario two parameters must be set; the file size and the arrival rate to the server. We have chosen to use a file size of 18 MB and a constant arrival rate of eight clients per second (these numbers are collected from the download statistics of the VLC media player [7]). In our simulations we use a simplified scenario where all the clients connect to the server within one single time frame, based on the arrival rate. For example, if there are 80 clients in total, the length of the time frame would be 10 seconds (number of clients divided by the arrival rate). The clients

have a constant arrival rate within this time frame. After the time frame no new clients connect to the server. The main choice of using a constant arrival rate is to simplify the experiment setup. It is easier to describe and relate to a constant arrival rate than a more advanced and realistic arrival time distribution. The single time frame is used to limit the running time of the simulations.

When simulating the behavior of a real network, some adjustments and simplifications must be done in order for the simulation to be practical. The constant arrival rate within a single time frame is such a simplification. In real-world file servers, the arrival time distribution is often modeled by a Zipf distribution or Poisson distribution (cf. Section 2.4). In addition, in file servers with a constant arrival rate, there is no time frame limitation. In Section 2.1, we discussed that the performance of CacheCast increases with the number of receivers. Thus, if we had used a distribution where the arrival times are more condensed, such as Poisson distribution, or had not used a time frame limitation, the number of connected clients on the server would increase. The end result would be increased performance gains from the CacheCast mechanism, due to the increased number of clients. Therefore, our simplified scenario is conservative in regard to the performance gains of the CacheCast file server.

This concludes the general introduction to the evaluations. In the next four sections we describe the experiments and analyse the results.

6.2 Rate control evaluation

The rate controller adjusts the transmission rate for each file session according to the rate of the currently fastest client. If the functionality of the rate controller is not correct, the other parts of the CacheCast file server which rely on it will not function optimally. Thus, this first evaluation is concerned with the behavior of the rate controller.

6.2.1 Experiment setup

The rate controller is described in Section 4.3. Its procedure of modifying the transmission rate is rather simple. If the currently fastest client could transmit a packet, increase the transmission rate (additive increase). If the packet was dropped by DCCP due to congestion, decrease the transmission rate (multiplicative decrease). Thus, the expected behavior of the transmission rate when measured over time is that it either increases or decreases each time it is updated. The rate controller also measures the fastest client. The expected behavior throughout a simulation is that the rate controller adapts to the currently fastest client. Both the transmission rate modification and the adaption to the currently fastest client is evaluated in this section. The evaluation is performed to verify that the rate controller functions according to the design.

Two experiments are conducted in this evaluation of the rate controller. In both experiments the transmission rate over time is measured. In the first experiment we study the rate modification and in the second experiment

the adaption to the currently fastest client is examined. The measurements are plotted in a graph and analysed based on the expected behavior. The metrics used in these experiments is the transmission rate measured in bits per second. In both experiments we study how the transmission rate varies over time.

A single simulation setup is used for these two experiments, with the topology and workload explained in the Section 6.1.2. The downlink speed of each client is set based on the distribution in Table 6.1. The number of clients in this simulation is 100. This number of clients is chosen based on the downlink speed distribution, such that each downlink speed group contains at least a couple of clients and the number of clients in each downlink speed group is equal to the distribution share value.

6.2.2 Results and analysis

In this evaluation part the simulations have been run multiple times to verify a similar behavior, but the results have not been averaged. The reason for not averaging is to visualize how the transmission rate behaves over time. The graph of the transmission rate over time would be a little different for each unique simulation. An average over all the graphs would not be useful to study how the transmission rate varies over time.

Transmission rate modification

For the first experiment - the study of how the rate controller is modified - we do not plot the transmission rate over time for the whole simulation. This would not be feasible in order to study the details of the rate variation. Instead, we present a small extract of the transmission rate. This extract shows the typical behavior of the variation in transmission rate found throughout the simulation. In Figure 6.2 the extract is depicted. Since, we have designed and implemented two transmission procedures, namely transmission of original blocks and transmission of encoded blocks, the experiment has been performed for both transmission procedures. The figure includes the results for each experiment. In both graphs the transmission rate both increases and decreases. However, for transmission of encoded blocks the transmission rate *either* increases or decreases, while for transmission of original blocks the rate remains constant in multiple time intervals. If we compare this behavior with the rate control algorithm (Formula 4.1) explained in Section 4.3.2, the transmission rate should either increase of decrease based on the status of the last sent packet, but not stay constant. Thus, when transmission of encoded blocks is used, the rate follows this behavior, but not when transmission of original blocks is used.

Investigation on the issue of a constant transmission rate, has revealed that in the intervals with constant rate, no block is being forwarded to the fastest client. However, blocks are selected and transferred to other clients, but not to the fastest one. This scenario happens because the currently fastest client has already received the blocks selected for transmission, and therefore it is not added to the set of clients to which the block is sent.

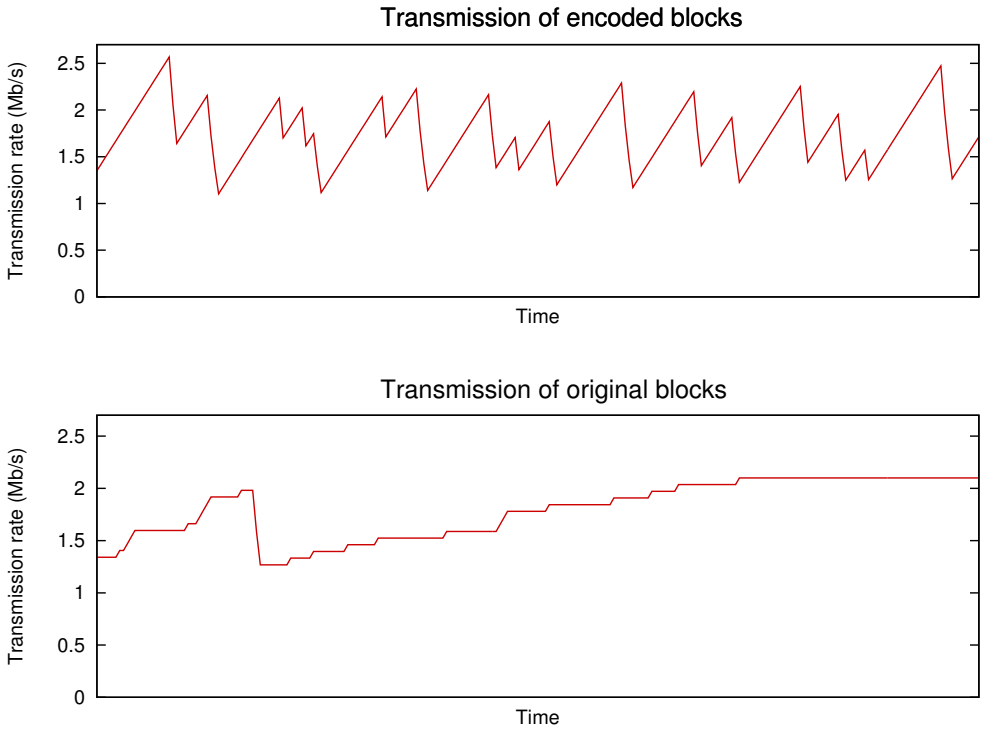

Figure 6.2: An extract of the transmission rate over time

This scenario is possible because the Round Robbin algorithm is used for block selection. In this algorithm, after the last block of the file is selected for transmission, the counter wraps around and the first block of the file is selected next. When the algorithm wraps around, we call this the start of a new *round*. If the client did not receive all file blocks during the first round, the server continues transferring blocks to the client during the second round. However, some blocks have already been received during the first round and these blocks are not transferred again. In cases where this happens with the currently fastest client, we have the issue in Figure 6.2.

To cover this scenario we create a new transmission attempt status called NOT SENT. In Table 6.2 we extent Table 4.2 with this new status and its description.

In Figure 6.3 we visualize the issue further using a flow diagram of the transmission procedure. There are now three possible outcomes of a transmission with msend(). For the two transmission attempt statuses SENT and DROPPED the rate controller gets feedback from DCCP. However, if the client has already received the selected block, the server does not send the block to this client. Since no packet is forwarded to DCCP, no feedback is returned. The rate controller depends on the feedback from DCCP to function, but in this scenario it does not get any feedback. The transmission rate should adapt to the currently fastest client, so if this scenario happens for this client, the transmission rate can not be updated, i.e. we have the behavior is Figure 6.2. Our tests show that this happens frequently.

Table 6.2: Updated list of packet transmission attempt statuses

Status	Description
SENT	The packet was successfully passed to the network layer by DCCP
DROPPED	The packet was dropped by DCCP because of congestion.
NOT SENT	The packet is not sent from the application because the client has already received the block contained in the packet. Since the packet has not been forwarded to the DCCP protocol, no feedback is reported back.

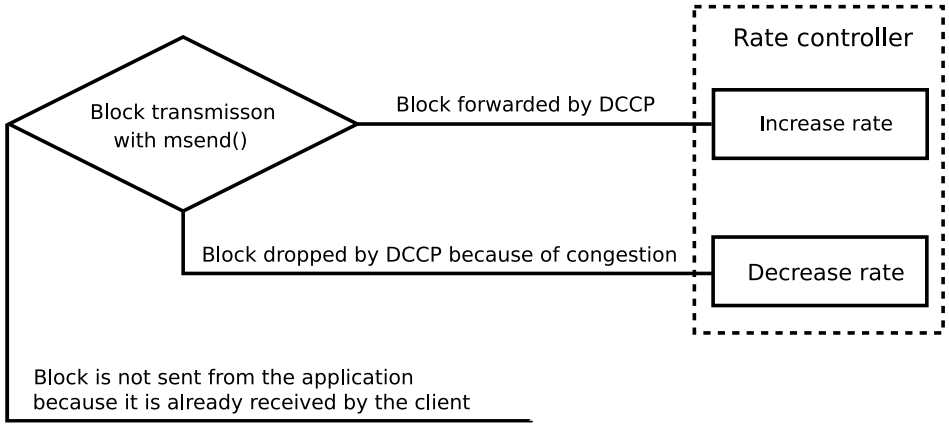

Figure 6.3: Visualization of the transmission attempt statuses

The issue with the NOT SENT transmission attempt status is not easily fixed. It is a consequence of using transmission of original blocks with the Round Robbin algorithm. When using transmission of encoded blocks, this scenario does not happen. The reason is that every client can receive any encoded packet, as explained in Section 3.6.2. The server always tries to transmit to all clients and the rate controller gets feedback for every client. Therefore, the NOT SENT status will not occur with Fountain codes.

From this discussion we conclude that the rate controller is not able to correctly modify the transmission rate when transmission of original blocks is used. It is a problem because it can not be assured that the transmission rate is updated, or at least updated according to the design of the rate controller. In later evaluations we set the transmission rate to a static value when transmission of original blocks is used in the simulation. This is by no means optimal, but in order to evaluate other aspects of the file server for both transmission procedures we use a static rate. It is set to the downlink speed of the fastest group of clients. According to Table 6.1 this value is 5 Mb/s.

Adaption to the currently fastest client

On the basis of the results of the previous experiment, we concluded that transmission of original blocks can not be used together with the rate controller. Therefore, in this experiment, where we evaluate whether the rate controller actually adapts to the currently fastest client, we do not run tests using transmission of original blocks, only using the Fountain codes approach.

In the previous experiment we did not present the transmission rate throughout the whole simulation. However, in this experiment, we plot the transmission rate from the beginning of the simulation to the end. This is done in order to observe how the transmission rate varies over time. The clients' downlink speed is distributed based on the distribution in Table 6.1. The expected result for the fastest client adaption is therefore that the transmission rate adapts to these speeds during different parts of the simulation. A fast client will download the file faster that a slow client. So it is expected that the transmission rate is high in the beginning of the simulation and becomes lower as the fastest clients are finished downloading.

In Figure 6.4 the results of the simulation is depicted, i.e. a graph of the transmission rate throughout the whole simulation. In addition, we plot the downlink speed of the currently fastest client. From the figure it is clear that the transmission rate changes during the simulation. If we compare the values of the transmission rate throughout the simulation to the downlink speeds in Table 6.1, there are clear indications that the transmission rate adapts to the client downlink speeds during different parts of the simulation. These indications are supported by the plot of the currently fastest client's downlink speed throughout the simulation.

In the investigation of this behavior, we have measured the time when the last client of each downlink speed group finished downloading. The

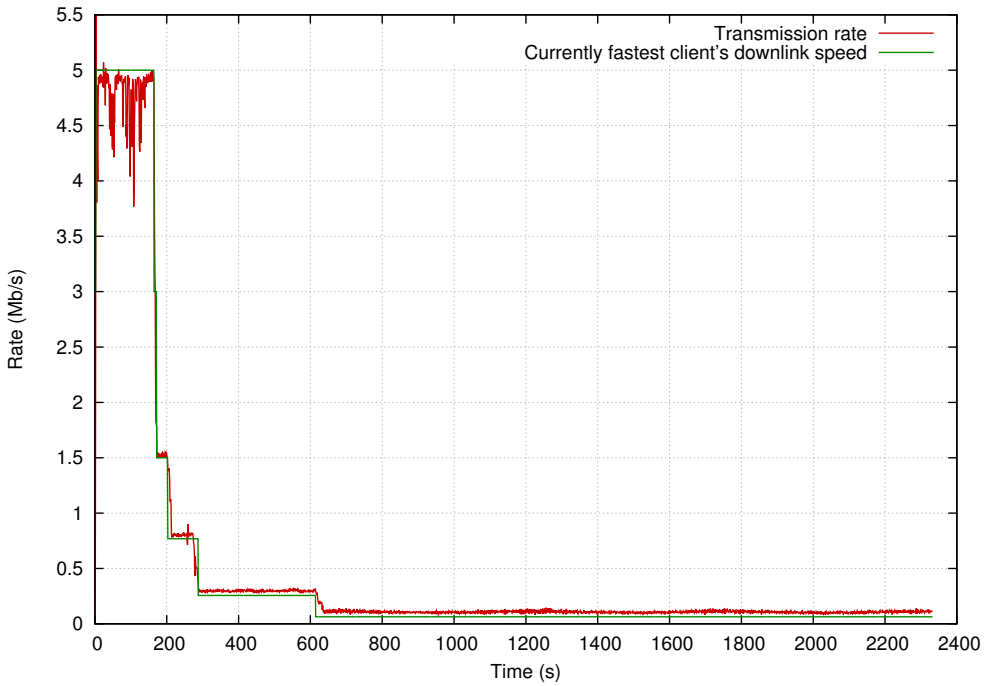

Figure 6.4: Transmission rate throughout the whole simulation

Table 6.3: Last finished download per downlink speed group

Downlink speed (kb/s)	64	256	768	1500	3000	5000
Time (s)	2338.4	622.7	269.3	198.6	161.9	157.8

measurements are given in Table 6.3. These numbers and the graph in Figure 6.4 correspond well. Whenever the last client of a downlink speed group is finished, the transmission rate adapts to the new currently fastest client, which belongs to a slower downlink speed group. For example, the last finished download for the 1.5 Mb/s downlink speed group is after 198.6 seconds. At this point in time the graph shows a clear decrease of the transmission rate. It drops to about 0.7 Mb/s, which corresponds to the slower downlink speed of 768 kb/s.

The time periods, in which the transmission rate adapts to a certain downlink speed, have different lengths. This stems for the well known fact that it takes more time for a slow client to download a file, than a fast client. However, the period with transmission rate of approximately 5 Mb/s is longer than expected when compared to the other periods. We attribute this to the fact that even though the transmission rate adapts to the clients with 5 Mb/s downlink speed, all the order clients are also receiving data during this period. Hence, the bandwidth consumed on the first hop link is not only due to the transmission of data to the 5 Mb/s clients. This also has the effect that the periods with transmission rate of 3 Mb/s and 1.5 Mb/s are short, since the clients with these downlink speeds have received most of the file during the 5 Mb/s period.

To summarize, this evaluation part shows that the Rate controller does not function according to the design when using transmission of original blocks. This is due to the fact the server does not transmit a block to a client which has already received it. When using transmission of encoded blocks, the Rate controller does function according to the design and it adapts the transmission rate to the currently fastest client. Whenever the last client with a certain downlink speed finishes downloading, the transmission rate drops and adjusts to the rate of the lower downlink speed group. In the next section we continue to study how the file download for clients with different downlink speeds is affected by the CacheCast support in the file server. Specifically we measure the amount of time it takes to download a file for clients with different downlink speeds.

6.3 Download time evaluation

A simple way of measuring the performance of a file download from a user perspective, is to measure the total time it takes to download the file. The download time of a file is essentially what really matters for the user. Therefore, in this evaluation we study the download times experienced by the clients of a CacheCast file server. We examine the behavior when multiple clients are downloading the same file from a single server. The clients have different downlink speeds and we study how this affects the total download times. The download times when using the CacheCast file server are compared against the download times when using FTP. As discussed in Section 3.1, FTP uses TCP for the data transmission part of the protocol. Therefore, in this and the following sections, when comparing the performance of the CacheCast file server to FTP, the comparison is not restricted to the specifics of FTP. Other protocols using TCP and which have similar functionality as FTP, could also have been used in the comparison. Generally, we compare the functionality of the CacheCast file server to the functionality of TCP.

The expected result of this evaluation is that the CacheCast file server performs better than FTP when there are multiple concurrent clients.

6.3.1 Experiment setup

The setup of the experiments is similar to the ones in the previous evaluation. The topology and workload is the same, and the clients downlink speeds are chosen from the distribution in Table 6.1. 100 clients connect to the server and download the same file. The same arguments as before applies for the number of clients in the simulation. We study the system when the network (first hop link) is congested. 100 clients with the selected downlink speed distribution consume enough bandwidth to create congestion in the network.

In this evaluation, we perform the experiments using both the CacheCast file server and FTP. For the experiment using the CacheCast file server, we use both transmission of original blocks and transmission of encoded blocks. Hence, three experiments are performed. When using transmission

Table 6.4: Download time when using a single TCP connection

Downlink speed (kb/s)	64	256	768	1500	3000	5000
Download time (s)	2491.0	623.5	208.7	106.7	53.7	32.6

of original blocks, we set a static transmission rate of 5 Mb/s, since the previous evaluation part showed that the rate controller does not function correctly for this transmission procedure. The metric in this evaluation is the total download time for a file. The download time is the time between a client connects to the server until it has received the last block of the file. The download time is measured for each client in the simulation. In order to compare the download times for each downlink speed group, all download times are arranged based on the downlink speed of the client.

6.3.2 Results and analysis

The results of the three experiments are depicted in Figure 6.5. The results reside in the same figure to easily compare the three experiments. The graphs are cumulative distribution functions of the download times for each client in the simulation. The graphs show the download time for all clients batched by downlink speed groups. For easy comparison of the different downlink speeds, we compute a relative download time. First, we measure the time it takes to download the file using a single TCP connection, when there are no other clients downloading the file. We call this download time "single TCP time". This "single TCP time" is measured for each downlink speed. These measurements are listed in Table 6.4. Second, the actual download time for each client is divided by the "single TCP time", giving a relative download time. Each client's download time is divided by the "single TCP time" corresponding to its downlink speed. For instance, if the download time for a 5 Mb/s client is 65.2 seconds, it is divided by 32.6 seconds, which gives a relative download time of 2. This means that the download time is 2 times higher than the download time if the client was alone on the server.

We give an example of how the graphs should be read. In the middle graph in Figure 6.5 the download times of the 5 Mb/s clients span from approximately 9 to 14. This means that for the 5 Mb/s client with a relative download time of 9, the download lasted 9 times longer than the "single TCP time". The last 5 Mb/s client received the file 14 times slower than it would have if it was alone on the server. From the graph we can also read the percentage of clients which experience less download time than a certain value. For example, the download time for around 50 % of the 5 Mb/s clients in the middle graph were less than 10 times the "single TCP time".

As expected, the graphs show that the download times are significantly decreased for the CacheCast file server compared to the FTP server. The results for the FTP server corresponds well with how FTP handles multiple clients; the same data is transferred multiple times. Therefore, the download time is related to the number of clients downloading the file. The download

time when using the CacheCast file server is reduced due to the introduction of the CacheCast mechanism in the network. Since equal packet payloads only traverse a link once, the transmission time of the packets from the server to the client is decreased.

In all graphs in Figure 6.5 the download time for the clients in the different downlink speed groups increases with the value of the downlink speed. For example, for the FTP server the download times for the 3 Mb/s client range from approximately 24 to 29, while for the 5 Mb/s client the span is from 38 to 48. This is related to the speed of the clients and the congestion on the bottleneck link. A fast client is more affected by congestion than a slow client, since it is not able to utilize its spare bandwidth capacity. A slow client may utilize its full bandwidth capacity, so the download time for this clients is not influenced by the congestion on the bottleneck link.

In both experiments using the CacheCast file server, the download time is significantly reduced compared to the download time for the FTP server. The shortest download times is achieved with transmission of encoded blocks. For the 5 Mb/s clients, the download time in this experiment is reduced by a factor over 10. This explicit factor depends on the specific experiment setup in this evaluation. The actual performance gains for different scenarios depend on the utilization of the CacheCast mechanism. In Section 6.6 we discuss this further.

The download times when using transmission of original blocks is also reduced compared to FTP, but not as much as when using transmission of encoded blocks. To investigate this issue we measure the download progression over time for a single client. We choose to examine a 5 Mb/s client since the effects seems to increase with the downlink speed. The graph of the download progression is depicted in Figure 6.6. The graph shows the percentage of remaining file blocks over time. The vertical dashed lines represents the start of a new round. The Round Robbin algorithm starts selecting blocks from the beginning of the file. When it reaches the last block, it starts from the beginning again, which is the start of a new round. The plot shows that the client starts receiving data fast. Immediately after the start of each new round there is a larger and larger flat section. After 240 seconds only 1 % of the file blocks remains. However, the client finishes the download after 444.8 seconds. The last 1 % of remaining blocks takes over 200 seconds to download.

To further investigate this behavior we plot only the download progression for the last percent. This plot is depicted in Figure 6.7. In this figure it is clearly visible that the graph remains constant over long periods of time, up to a whole round. During these time intervals the client does not receive any data. It just waits for data to arrive. This issue drastically decreases the performance, and its cause is related to the NOT SENT transmission attempt status described in the previous section.

If a client has already received the block which is selected for transmission, the server does not transfer any data to this client. When this happens for subsequent blocks, no data is transferred to the client during multiple transmission attempts. For example, if a client has already received the first 50 blocks of the file, each new round the client just waits during the trans-

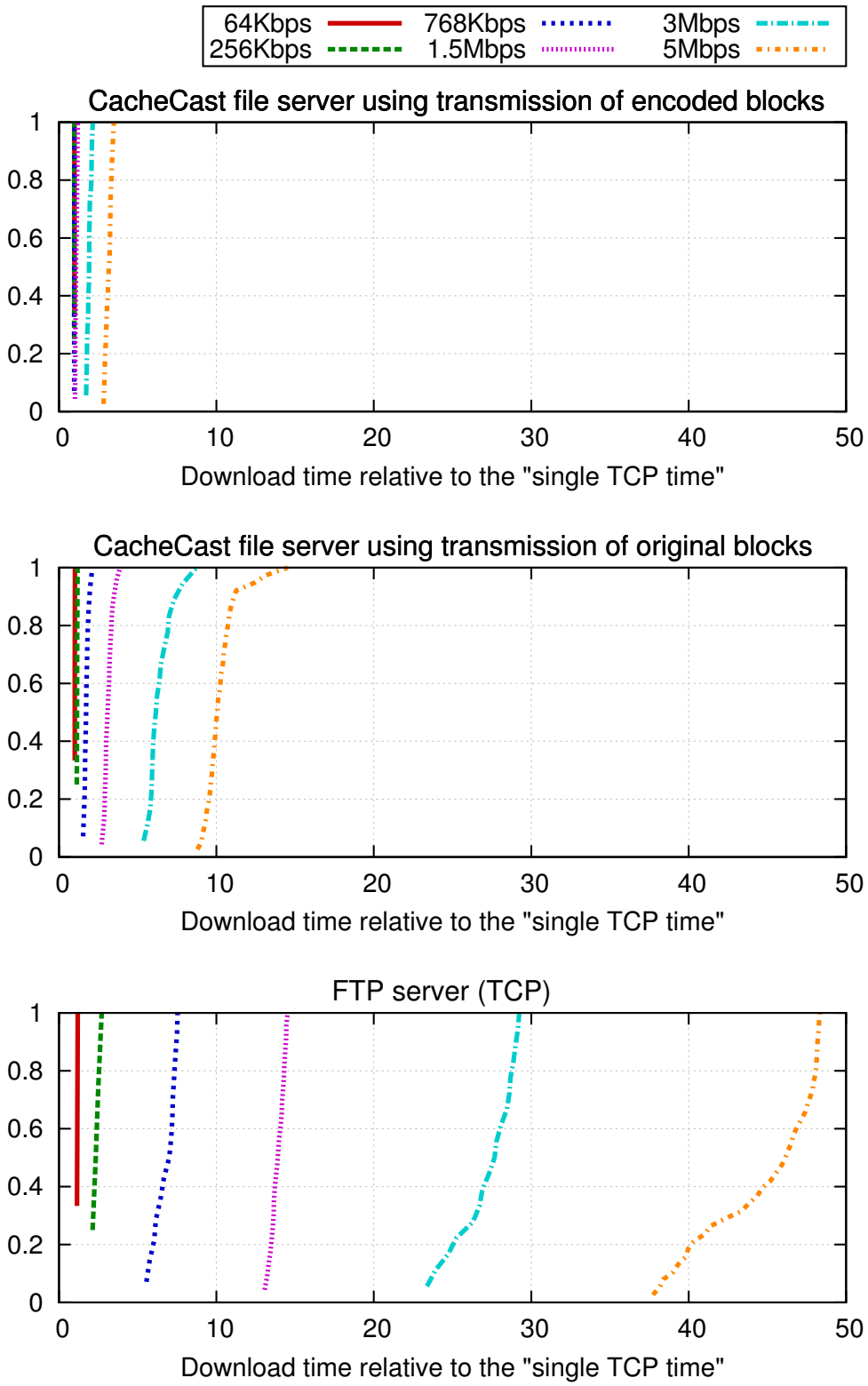

Figure 6.5: Download times for the CacheCast file server and the FTP server

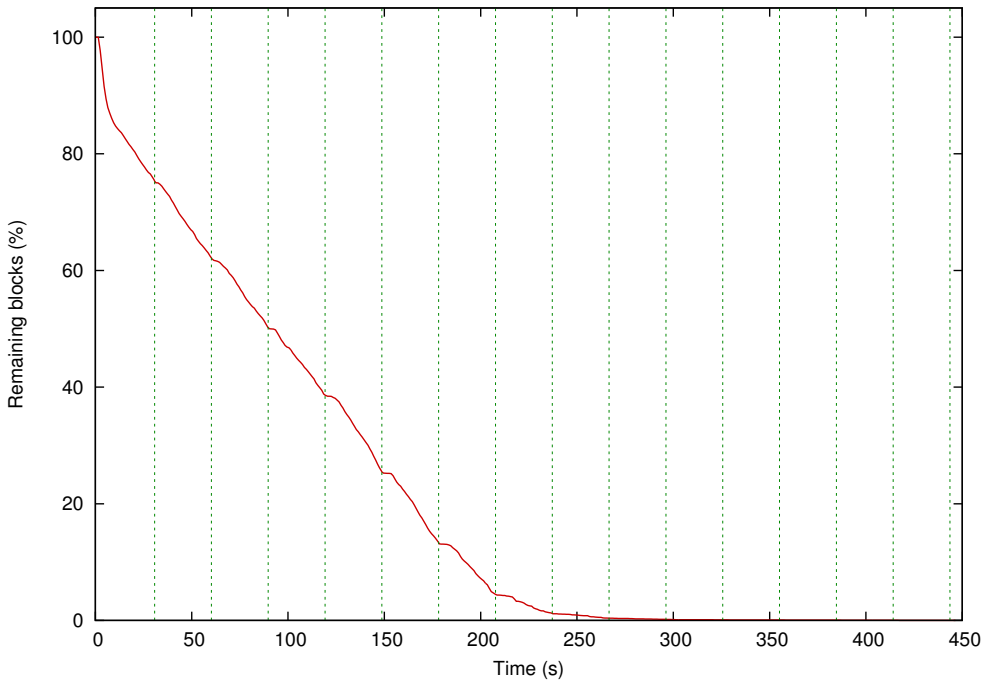

Figure 6.6: Download progression for a single 5 Mb/s client

mission of the first 50 blocks. This behavior is suboptimal, and it is due to the core functionality of the block selection. If a block has already been received by a client, there is no point in transferring it to the client again. Since only one block is transferred to a subset of the connected clients, it may happen that clients don't receive blocks for long periods of time, as Figure 6.7 shows. Our tests have revealed that this issue happens frequently.

For transmission of encoded blocks, the issue it not present. The server will always try to transmit data to all clients. Even though transmission of original blocks performs better than FTP, the discussion in this section and in the previous one shows that it has issues both related to the rate controller and the transmission procedure. In Section 8.2 we propose some research directions in which to cope with these issues. In the following evaluations we perform experiments only using transmission of encoded blocks.

To summarize, the CacheCast file server decreases the download time considerably when comparing it to the download time of an FTP server. The decrease is a consequence of the introduction of the CacheCast mechanism in the network. When continuing with the next evaluation we shall see another impact of the CacheCast support in the file server.

6.4 Bandwidth consumption evaluation

The CacheCast mechanism removes redundant data transfers on links in the network. As a consequence, the download time of a file is reduced, as discussed in the previous evaluation. Another direct consequence of removed payloads is that less bandwidth is consumed on each CacheCast

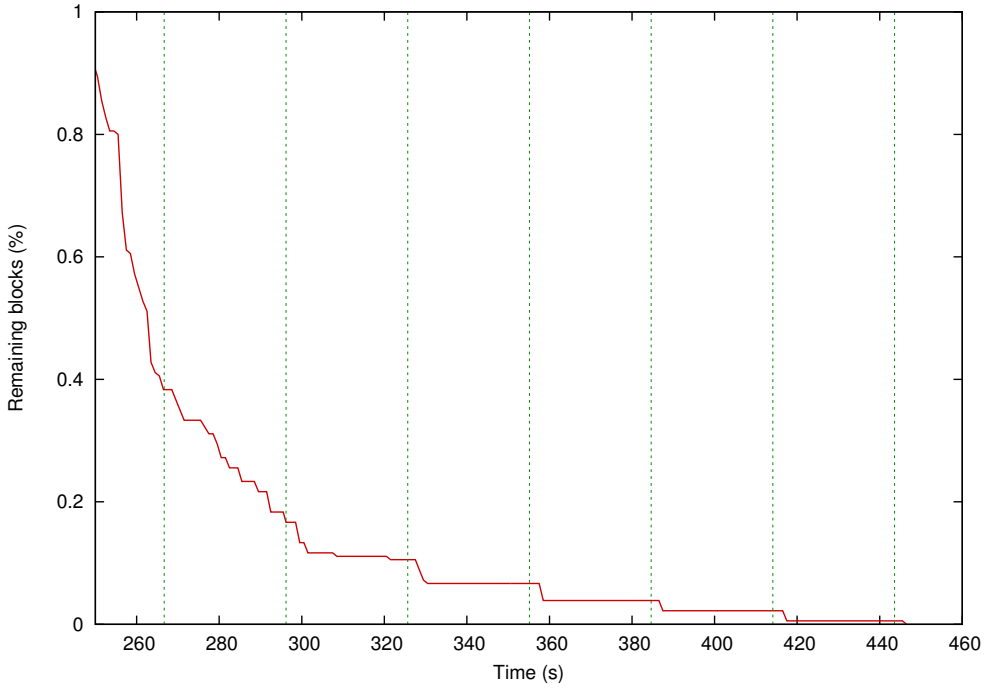

Figure 6.7: Download progression for the last percent of remaining blocks

supported link. In this evaluation we explore how the introduction of CacheCast support in a file server affects the bandwidth consumption in the network.

This evaluation is very related to the previous ones, especially the rate control evaluation. The rate controller adjusts the transmission rate, and this rate affects how much data is transferred onto the link per time unit. When more data is transferred the bandwidth consumption increases. The bandwidth capacity on the bottleneck link sets an upper limit on the amount of data transferred through the network per second.

6.4.1 Experiment setup

The experiment setup in this evaluation is the same as in the previous evaluations. This is done to easily compare the results of the different evaluations. In this evaluation we do not consider transmission of original blocks, due to the aforementioned issues concerning this transmission procedure. Thus, the experiments in this evaluation is performed using the CacheCast file server with transmission of encoded blocks and an FTP server. The results of these experiments are compared and analysed. The metric used is the bandwidth consumption in percent on the bottleneck link. The expected outcome of the experiments is that the CacheCast file server consumes less bandwidth than the FTP server on the bottleneck link.

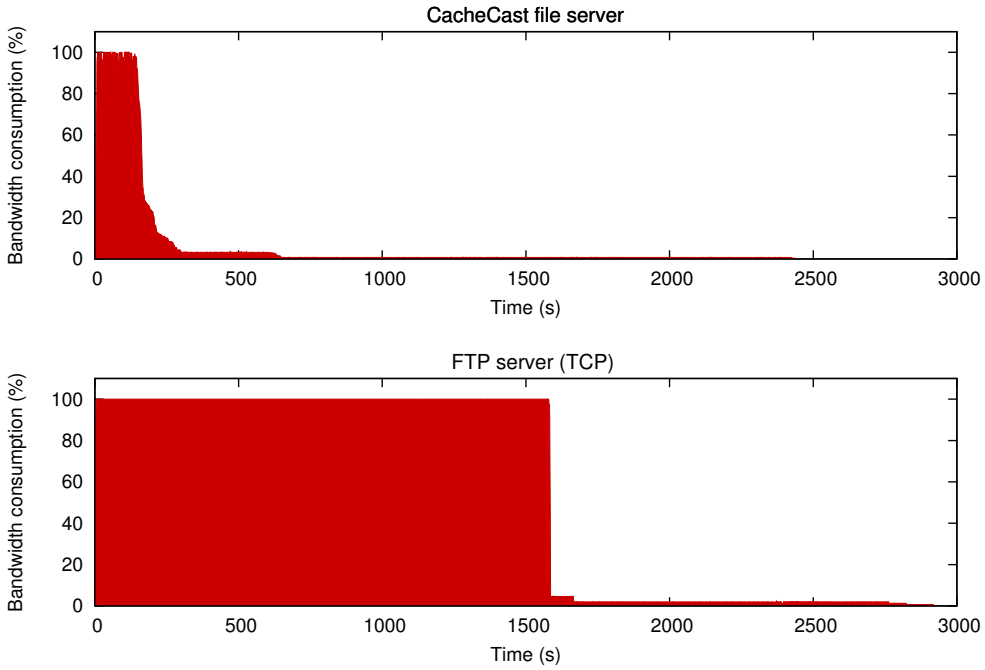

Figure 6.8: Bandwidth consumption on the bottleneck link

6.4.2 Results and analysis

In the experiments performed in this evaluation we measure the bandwidth consumption on the bottleneck link per second throughout the whole simulation. The measurements are presented in percent of the total bandwidth capacity on the bottleneck link. The results for both the CacheCast file server and the FTP server is depicted in Figure 6.8.

The figure shows a significant decrease in bandwidth consumption for the CacheCast file server when compared the FTP server. If we compare the top graph in this figure to Figure 6.4, the graphs have approximately the same shape. Thus, the bandwidth consumption is governed by the transmission rate. However, the bandwidth consumption is higher than expected in some parts of the graph, when compared with the value of the transmission rate throughout the simulation. For example, in the first 150 seconds the transmission rate stays at approximately 5 Mb/s (cf. Figure 6.4), while the bandwidth consumption is at 100 %, which corresponds to 10 Mb/s. This behavior is related to how the Scheduling module schedules the packet transmissions. Formula 4.2 is used to calculate when a new packet transmission should occur. This formula takes into account the size of a single packet and the Scheduler schedules the transmission accordingly. However, when transmitting to multiple clients, there are a number of headers in the packet train which are also transferred onto the link. This increases the amount of data transferred from the server and consequently the bandwidth consumption in the network rises. The Rate controller adjusts the transmission rate to the speed of the fastest client, not to the bandwidth capacity of the bottleneck link. Even though the packet transmissions are scheduled based on the transmission rate, the transmission rate does not

give the exact amount of data transferred onto the link per second.

The top graph in Figure 6.8 shows that the CacheCast file server consumes 100 % of the bottleneck link bandwidth for approximately 150 seconds and then the bandwidth consumption decreases and stays constant in different intervals throughout the simulation. The last client is finished downloading after 2446 seconds. The FTP server consumes 100 % of the bandwidth capacity for approximately 1600 seconds and then it quickly decreases to around 2 %. The last client when using an FTP server is finished after 2918 seconds. The bandwidth consumption for the CacheCast file server is approximately 12 % of the consumption when using an FTP server. Our simulations show that the average packet train length is 14. In our simulations the packet payload size is 1024 bytes and the header size is 50 bytes. Calculations on the average packet train length show that the number of bytes sent onto the link for the CacheCast file server is around 11 % of the number of bytes when using an FTP server.

6.5 Fairness evaluation

In the previous evaluations, all clients are downloading the same file from the file server. However, the CacheCast file server is able to handle multiple files. This means that clients can download multiple files concurrently. Therefore, we study in this evaluation how the CacheCast file server treats multiple groups of clients downloading different files.

When multiple files are being downloaded concurrently, a factor to consider is the fairness between concurrent downloads. The server should ensure that each client get an equal share of the bandwidth capacity. The CacheCast file server uses DCCP, which ensures fair bandwidth share among concurrent data streams in the network. The CacheCast mechanism does not impact the end-to-end fairness [43]. As a consequence, the fairness in the network between the different downloads is already preserved. The transmission rate from the server is governed by the rate controller. As previously explained, the rate controller uses the feedback from DCCP to modify the transmission rate and adapts it to the currently fastest client. In the Rate control evaluation we demonstrated the correctness of the Rate controller according to the design. However, we did not evaluate how the transmission rate affects the fairness among the client downloads. In this section, we perform experiments to examine whether the CacheCast file server preserves the fairness among the clients downloading the same file and between groups of clients downloading different files. Since, the transmission rate is adapted to the DCCP congestion control, which preserves the fairness among the concurrent file downloads in the network, we expect that the Rate controller preserves the fairness of DCCP and that the CacheCast file server fairly shares the bandwidth capacity for the clients.

6.5.1 Experiment setup

The experiments performed in this evaluation differ from the previous experiments in a number of areas. In this evaluation we study the impact

of clients downloading different files from the server. Therefore, all clients do not download the same file anymore. In the simulations we use two groups of clients, each downloading a different file from the CacheCast file server. To study how the fairness is affected by the total number of clients, we perform experiments with 10 and 200 clients in total. In order to test the system with many clients, a higher number than 200 could be used, but 200 is chosen to limit the running time of the simulation.

The size of each group of clients is varied to examine the fairness between differently sized groups. For the experiments with 10 clients in total we perform five experiments using the following group sizes; 5 and 5, 4 and 6, 3 and 7, 2 and 8, 1 and 9. When there are 200 clients in total, the group sizes are; 100 and 100, 80 and 120, 60 and 140, 40 and 160, 20 and 180.

To easily compare the bandwidth share, we do not use the downlink speed distribution in Table 6.1, as in the previous experiments. Instead, each client has the same downlink speed of 10 Mb/s. This specific speed is chosen such that in all group size configurations (even when a group only contains a single client) a single group is able to consume all the bottleneck bandwidth capacity. Then, we can compare the bandwidth share between two groups of any size.

The metric used in this evaluation is the share of the bandwidth capacity between the two groups of clients on the bottleneck link. In addition we measure the end-to-end throughput for the two groups, in order to compare it against the bandwidth share. All measurements are taken when the simulation is in a steady state, i.e. no clients are arriving or leaving the CacheCast file server. This is done to study the fairness between the groups when the group size remains constant. This makes it easier to compare the different experiments, and the effects of clients arriving or leaving are ignored.

6.5.2 Results and analysis

The measurements in this evaluation are collected by running multiple simulations with the abovementioned setup. From the measurements we have calculated an average and the standard deviation. When running the first simulations, we experienced an unrealistically large standard deviation for the throughput. Investigation on this issue revealed that be problem was related to the order of the packets transferred with msend() and the output queue on the server. The output queue is a standard Drop Tail queue [18]. When a Drop Tail queue is full, it drops all new packets arriving. The clients, to which to transfer a block, have always the same order, which is the clients' arrival order to the server. If the output queue does not have enough room for the whole sequence of packets transferred with msend(), the last packets in the sequence are dropped by the queue. Since the clients have always the same order, the last packets in the sequence is always destined for the same clients. Thus, when the output queue is full, the packets destined for the same clients are always discarded. The clients toward the end of the packet sequence therefore experience a larger packet loss than the other clients. To overcome this issue, the order of the packet sequence transferred

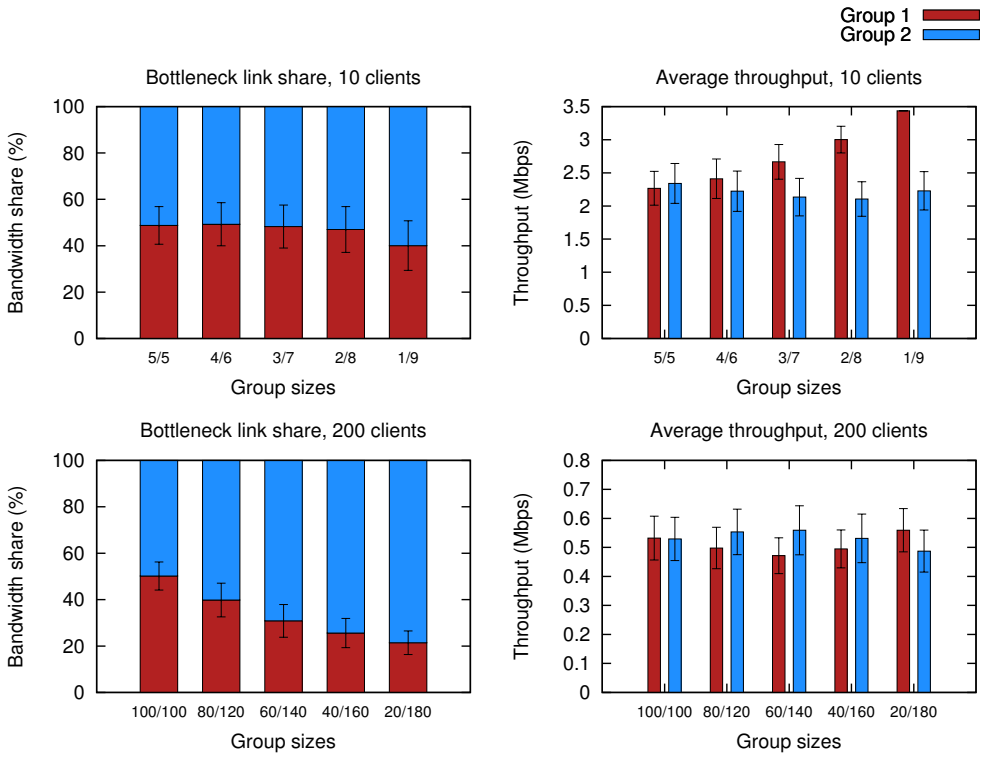

Figure 6.9: Bandwidth share on bottleneck link and end-to-end throughput

by msend() is altered. The vector of sockets which is input to the msend() function is randomly shuffled, so each packet sequence sent from msend() have a different order. By doing this, the dropping policy of the output queue does not always affect the same clients. The use of a *Random Early Detection (RED)* queue [32] could possibly fix this issue without the need of the random shuffling, but this has not been investigated in this thesis. However, in a real network there are no guarantees that all routers use the RED queueing policy.

The need for randomly shuffling the packets in a packet train is an important insight. The issue with the Drop Tail queue does not only affect the CacheCast file server, but it would affect any application using the CacheCast server support. Therefore, the random shuffling should be integrated in the msend() system call.

In Figure 6.9 we present the results of the experiments. The average bottleneck link share and the average throughput per client group is shown, together with the standard deviations. In order to easily compare the behavior of the system with 200 and 10 connected clients, both the bottleneck bandwidth share and the throughput are depicted in the same figure. The results are presented for the two groups of clients downloading different files from the server. The size of Group 1 is decreased in the different experiments, while the size of Group 2 is increased. When there are 10 clients in the simulations the bandwidth share stays at approximately 50 % for each group. The only exception is for groups sizes of 1 and 9. The larger groups gets around 60 % of the bandwidth capacity. The throughput in this

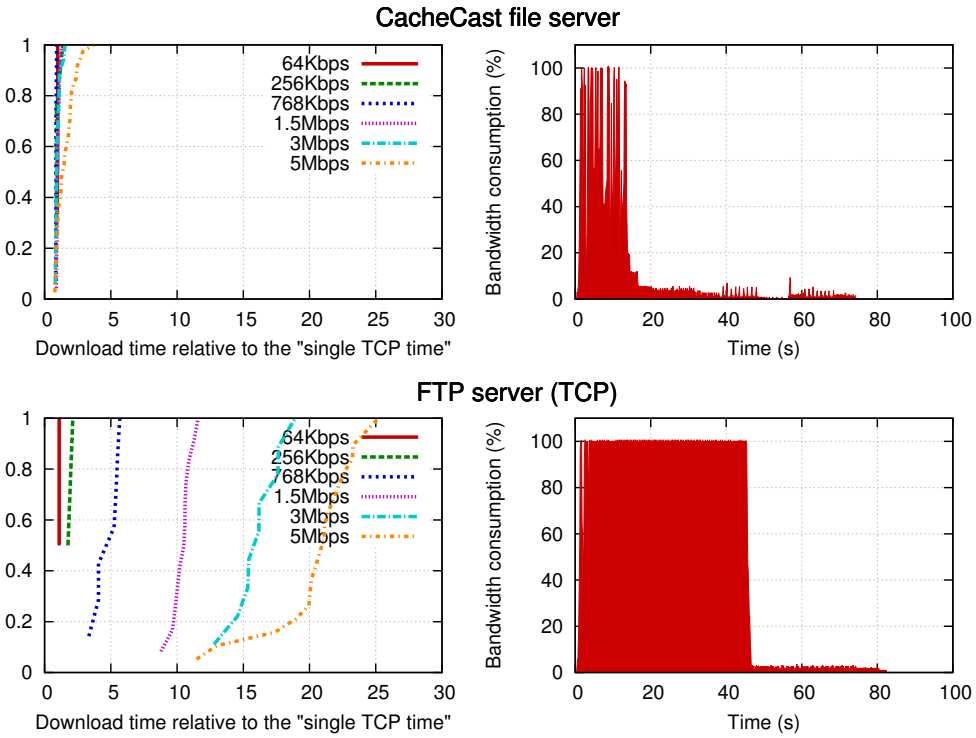

Figure 6.10: Download time and bandwidth consumption for 500 kB file

scenario increases for Group 1 when its size is decreased, but the throughput for Group 2 has approximately the same value even when the group size is increased.

For the experiments with 200 clients, when the ratio between the group sizes increases, the larger group gets more bandwidth capacity. Still, the ratio of bandwidth share does not fully comply with the group size ratio (the ratio between the size of the two groups). This is a results of the CacheCast mechanism. Due to the redundancy removal, only the first packet in a packet train carries the payload. Therefore, the ratio between the number of packets sent and the number of bytes sent is not equal. As a consequence, when the packet train length is short (as in the experiments with only 10 clients), the bandwidth share is not fully proportional to the group size ratio. However, when the packet train length increases, these two ratios will even out.

The graphs to the right in Figure 6.9 show, even though the ratio of bandwidth share is not fully proportional to the ratio between the groups, that the clients' average throughput is not affected by this difference. We therefore claim that the CacheCast file server preserves a fair share of network resources between clients, both inside the same group and across different groups.

6.6 More insights

The download time evaluation and bandwidth consumption evaluation show significant performance gains for the CacheCast file server compared

to the FTP server. We attribute these performance gains to the CacheCast mechanism. When there are multiple clients downloading the same file concurrently, CacheCast is able to remove redundant data transfers. In Section 3.6.3 we discussed that the performance of the CacheCast file server increases with the number of overlapping clients. In order for overlaps to occur, new clients must start the download of a file while there are already clients downloading the same file. From this fact we deduce that the amount of overlapping between downloads depends on both the request rate and the download time of the file. The download time depends on the file size and the download speed. Therefore, if the download time is short the request rate must be high, in order for overlaps to occur. If, on the other hand, the download time is long it does not require such a high request rate for overlaps to occur. Moreover, when either the request rate or the file size is increased, the amount of overlapping is increased, and when either value is decreased the amount of overlapping is decreased.

When the amount of overlapping changes, the amount of redundancy CacheCast is able to remove also changes. As an example, we have run the experiment in the download time evaluation and bandwidth consumption evaluation again, but the file size is decreased from 18 MB to 500 kB. The results are depicted in Figure 6.10. Even though CacheCast is still able to remove much redundancy (since the request rate is still high), the performance of the CacheCast file server has decreased compared to the FTP server. The decreased file size has resulted in less overlapping, so there is less redundancy for CacheCast to remove. The bandwidth consumption has increased from 12 % when using the 18 MB file to 21.5 % for the 500 kB file. The average packet train length has decreased from 14 to 6.6, thus the CacheCast file server is not able to benefit as much from CacheCast as in the original experiment.

In all previous experiments the bandwidth capacity of the bottleneck link is 10 Mb/s. Both the CacheCast file server and the FTP server are able to consume 100 % of the bandwidth capacity during different time intervals. The benefits observed in the previous experiments are shorter download times and less bandwidth consumption in the network. These benefits are seen in experiments with a congested bottleneck link. In order to show the benefits when the file server is not able to consume 100 % of the bottleneck link capacity, we run the same experiment as in the download time evaluation and bandwidth consumption evaluation again, but with a bottleneck link capacity of 300 Mb/s. This value is chosen to avoid a 100 % bandwidth consumption. The results are depicted in Figure 6.11. In this figure, the graph of the bandwidth consumption only contains the first 300 seconds, since in the rest of the simulation only the slowest clients are downloading and these consume about the same bandwidth for both the CacheCast file server and the FTP server. These results show that the download time is not much different for the two file servers, when there is no congestion in the network. However, the bandwidth consumption is significantly lower for the CacheCast file server. While the FTP server reaches over 95 % of the bandwidth capacity, the CacheCast file server never reaches 10 % of the capacity. Calculations show that the total bandwidth

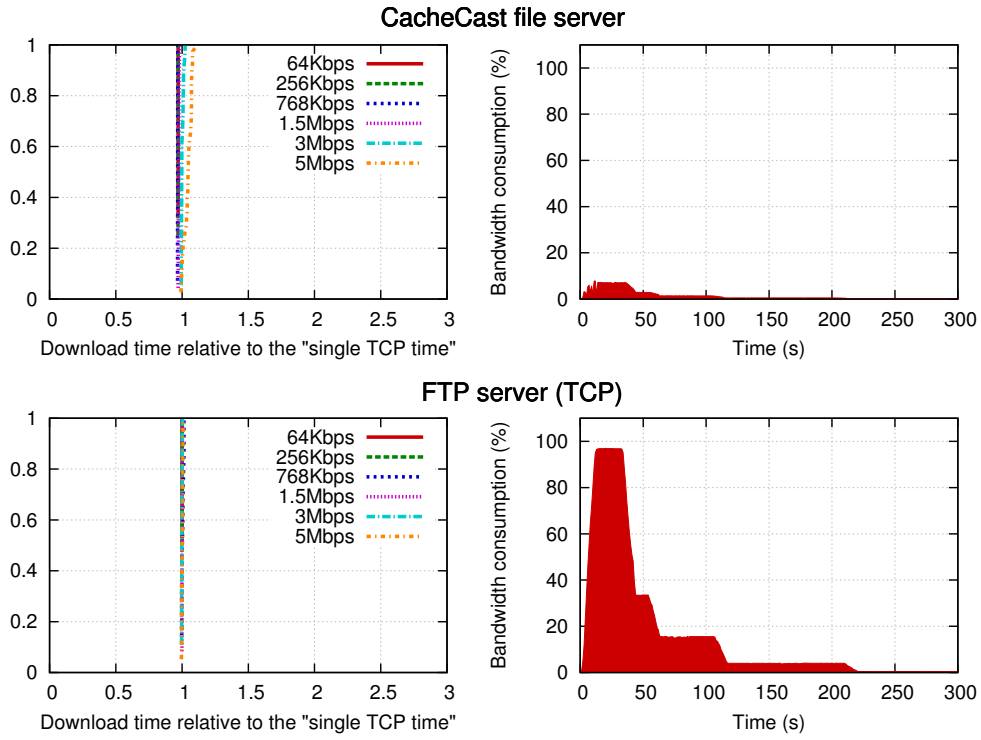

Figure 6.11: Download time and bandwidth consumption with a bottleneck link bandwidth of 300 Mb/s

consumption for the CacheCast file server is 9.8 % of the consumption for the FTP server. For an over provisioned network, the benefits of CacheCast is only related to the number of bytes transferred.

6.7 Critical discussion

The main focus of this thesis has been to investigate the network effects of a file server with CacheCast support. Due to this focus, we implemented the system in the ns-3 network simulator. As a consequence of this decision, we were not able to evaluate the computational complexity of the system. For the two transmission procedures, transmission of original blocks and transmission of encoded blocks, the source of the computational complexity is different. For transmission of original blocks, the complexity lies in the computation of the fastest client within the Rate controller and in the procedure of creating the set of clients to which to transfer a block. Since the evaluation revealed some issues with transmission of original blocks, we do not go into details on the analysis of its computational complexity.

More importantly, we examine the complexity of the procedure of transmitting encoded blocks. With this transmission procedure, the Rate controller still calculates the fastest client, but a new set of clients does not need to be calculated for each block transmission. By storing all clients in a priority queue, the computational footprint of finding the fastest client could be significantly reduced. Nevertheless, the most computationally intensive

tasks in the Fountain codes approach, are the encoding and decoding of the blocks. These tasks are neither implemented nor evaluated in this thesis. The complexity of the encoding and decoding depends on multiple factors, such as the specific coding algorithm, the file size, and the block size. Shojania and Li [42] have analysed the performance of different Fountain codes algorithms on multiple computing devices. Generally the encoding task requires the least computation and the encoding is performed on the server, which in most cases have higher computing power than the clients. In addition, the encoding procedure could run as an off-line job, creating precoded blocks. The critical computational workload is therefore the decoding on the clients. Shojania and Li show that an iPhone 3GS, which has a CPU speed of 600 MHz, is able to decode a 10 MB file, with a block size of 1024 bytes, in under one second. The download time of a 10 MB file for a 5 Mb/s client, alone on the server, is 16.4 seconds. Thus, in this scenario the decoding process takes less than 6.1 % of the time it takes to download the encoded blocks. Faster CPU's are able to decode much faster. However, the computational complexity increases with the file size.

In order to study the performance of the full system and to prove its feasibility, the above discussion shows that the computational complexity should be taken into account. However, this is beyond the scope of this thesis.

6.8 Summary

In this chapter, we have evaluated the performance of the CacheCast file server. The Rate control evaluation showed that when using transmission of original blocks the Rate controller is not able to correctly modify the transmission rate. However, when using transmission of encoded blocks the Rate controller functions according to the design. In Section 6.3 and Section 6.4 we compared the download time and bandwidth consumption of a CacheCast file server to an FTP server. In both evaluations the CacheCast file server performed significantly better than the FTP server. In the experiment performed, the download time for the 5 Mb/s clients is over ten times shorter for the CacheCast file server than for the FTP server, and the bandwidth consumption is reduced by approximately 88 %. The Fairness evaluation shows that the CacheCast file server shares the bandwidth capacity fairly between the connected clients.

The performance gains of the CacheCast file server are due to the deployment of CacheCast is the network. The magnitude of these gains is related to both the download time of the file and the request rate. When the number of concurrent clients increases, the magnitude of the performance gains also increases.

Chapter 7

Related work

The CacheCast file server is an efficient file server designed for single source multiple destination transfer. There has been much research addressing the design of a file transfer mechanism for efficient single source multiple destination scenarios. In this chapter, we present some work related to the CacheCast file server.

7.1 Multicast file transfer

Most former work in the area of single source multiple destination file transfer is based on multicast techniques, such as IP multicast [15]. IP multicast was developed to efficiently transfer data to multiple receivers. The unicast transmission scheme is based on host to host communication, while in multicast schemes, a host transfers data to a group of hosts[1]. In IP multicast the sender transfers the data only once and the network is responsible of replicating it to all receivers. This mechanism consumes much less bandwidth then when multiple unicast connections are used to transfer the data.

Some examples of reliable multicast file transfer protocols include; *Reliable Multicast Protocol (RMP)* [48], *Xpress Transport Protocol (XTP)* [8], *Fcast* [22] and *Pragmatic General Multicast (PGM)* [23]. RMP and XTP transfer original data and use sequence numbers to ensure reliability - the same functionality as in transmission of original blocks (cf. Section 3.6.1). Fcast and PGM use a Forward Error Correction (FEC) scheme to ensure reliable transfer. It is a similar mechanism as the Fountain codes approach used in transmission of encoded blocks (cf. Section 3.6.2).

While IP multicast is an efficient single source multiple destination transfer mechanism, it has not been widely deployed outside of individual autonomous systems due to problems with scalability, security and group handling (among others) [16]. In addition, IP multicast breaks the end-to-end relationship in the network which makes it difficult to employ end-to-end congestion control.

In order to overcome the limitations of IP multicast, another approach to multicast has been developed, called *Application Layer Multicast (ALM)* [26].

[1] The sender side could also consist of a group of hosts

ALM systems do not depend on specific network multicast support (like IP multicast), but may use it if it is available. Otherwise the multicast communication is achieved with multiple unicast connections. Overlay networks are often established on top of the existing network topology to distribute data more efficiently. However, since network multicast support in the Internet is limited, ALM systems has inherently the same limitations of using multiple unicast connections as described in this thesis. A successful system using the ALM approach is BitTorrent [39], which we discuss in the next section.

7.2 Block ordering

In Section 3.5 we explained that in order to achieve synchronous transmission in the CacheCast file server, the file blocks must be reordered before transmission. Since the client can correctly order the blocks after they have been received, this reordering can be safely done.

In TCP and in the multicast protocols RMP and XTP introduced in the previous section, the transmission order is sequential, but sequence numbers are used to correct reordered packets. The BitTorrent system relies on the same argument of reordering as the CacheCast file server. In BitTorrent, a file is located on multiple peers and the file is split into multiple file chunks. A user downloading a file, can receive the file chunks in any order and from any peer. The reception order of the file chunks is unknown, thus the file is correctly ordered on the client-side. The core principle in BitTorrent is to download from multiple peers to increase the performance. Without the fact that the block transmission order is irrelevant, this would not be possible.

Both the CacheCast file server and BitTorrent depends on the possibility of transferring the different parts of a file in any order. Another system similar to BitTorrent, called *Slurpie*, is described by Sherwood et al. in [41]. Both Slurpie and BitTorrent are designed to achieve the same goals as the CacheCast file server; increasing the download performance for large, popular files.

7.3 Client batching

In order to benefit from the CacheCast mechanism, the same data must be transferred to a batch of clients. The CacheCast file server reorders the blocks to achieve this batching.

Different batching techniques have formerly been developed to increase the performance of multicast transmission especially in Video-on-Demand systems. In the typical batching technique, clients with different arrival times at the server is grouped together. If a client requests some content from the server, the download is not started immediately. Instead the client is inserted into a set of clients where each client requested the same data. When a certain condition is met, all clients in the set are served together using multicast. This batching technique is explained by Dan et al. in [14].

An advantage of this kind of batching is that the clients are initially synchronized is time which enables efficient multicast transmission. However, the major disadvantage is that the clients will have to wait a certain amount of time. When compared to the method used in this thesis, where the clients do not need to wait, the arrival time batching is suboptimal.

Hua et al. proposed another technique of batching clients in a Video-on-Demand system called *Patching* [27], to overcome the waiting time limitation of conventional batching and to improve the multicast performance. In this technique, multiple streams of the same video file are combined when possible, by using extensive caching on each client. We illustrate the functionality of Patching with an example: Two clients have been receiving the same video stream for 2 minutes, we call this stream A. Then, another client want to stream the same video. This client starts receiving the video from the beginning. This stream is called B. In addition, the new client also receives data from stream A, which is stored locally in a cache. Whenever the client has received (on stream B) the segments of the video file preceding the first segment stored in the cache (which was received on stream A), stream B is discontinued and the client now receives data only from stream A. Stream B was used to *patch* the missing portions of the video file for the new client.

The way new clients are added to existing streams in the Patching technique is similar to how the CacheCast file server handles arriving clients. Whenever a new client requests a file which is currently being downloaded, it is added to the current data flow, and it starts receiving the same blocks as the other clients. As opposed to Patching, there is only a single data flow for the same file in the CacheCast file server. The Reliability module ensures that each client eventually receive all blocks.

7.4 Summary

The CacheCast file server contains elements from all the aforementioned techniques. It is designed for single source multiple destination transfer, as are the different multicast file transfer protocols we introduced above. Both the CacheCast file server and BitTorrent rely on the fact that blocks can be transferred in any order. In addition, in order for CacheCast to remove redundant data transfer, the CacheCast file server batches clients into groups, where the clients in a group are downloading the same file.

Even though the CacheCast file server has functionality similar to other systems, the CacheCast file server as a whole is unique. It is specifically designed to include support for CacheCast, the newly developed mechanism of removing redundant data transfers from single source multiple destination transfers.

Chapter 8

Conclusion

In this thesis, we have designed, implemented and evaluated the CacheCast file server. The CacheCast support is added to the file server to remove redundant data transfers when there are many clients downloading the same file concurrently.

In Chapter 2, we provide the necessary background information to design and implement the file server. While support for CacheCast can be easily added to a live streaming system, CacheCast support in a file server requires a special system design. This design is presented in Chapter 3. We describe the issue of multiple clients downloading the same file concurrently and illustrate how CacheCast can optimize the transmission in the overlapping time periods, by reordering the blocks before transmission. Two transmission procedures are presented, namely transmission of original blocks and transmission of encoded blocks. For both of these procedures we discuss how to meet the main requirement for file transmission, i.e. reliability.

In Chapter 4, we explain how the basic design considerations are transformed into a system architecture. Three necessary modules for the CacheCast file server are identified; Reliability module, Rate control module, and Scheduling module. The implementation of these modules in the ns-3 network simulator is described in Chapter 5. In that chapter we also explain the implementation of the CacheCast mechanism in ns-3.

Chapter 6 contains the description of the evaluation performed in this thesis. Four aspects of the CacheCast file server are evaluated; the Rate controller, the download time experienced by the clients, the bandwidth consumption in the network, and the fairness among the different clients. In Chapter 7, we discuss some related work for the CacheCast file server.

8.1 Results

The main goal of this thesis is to add CacheCast support to a file server, in order to remove redundant data transfers from popular downloads. In addition, we have evaluated the system and based on this evaluation we can state the feasibility of the CacheCast file server.

The first achievement of this thesis is the design of the CacheCast

file server. A thorough study of file transmission and an analysis of the requirements for a file server with CacheCast support has led to the system design and CacheCast file server architecture explained in Chapter 3 and Chapter 4. The CacheCast file server is able to utilize the CacheCast mechanism by reordering the blocks before transmission and transmitting the same block to multiple clients.

However, the design and implementation of a system is not enough to demonstrate its feasibility. Therefore, a performance evaluation of the CacheCast file server has been performed. The results of this evaluation show significant performance gains for the CacheCast file server compared to an FTP server. These gains are related to the bandwidth consumption in the network and the download time experienced by the clients. When a popular file is downloaded from the CacheCast file server, great bandwidth savings are achieved. In addition, all clients receive the file significantly faster than when using an FTP server. The fairness evaluation shows that the CacheCast file server preserves a fair share of the bandwidth capacity in the network among the connected clients.

The design and evaluation of the CacheCast file server demonstrate that CacheCast support can be successfully added to a file server. The CacheCast file server is an efficient file server which saves network resources. It is a feasible alternative to the standard FTP server.

When running the experiments in the fairness evaluation we experienced an issue with the order of the packets transferred with msend() and the Drop Tail output queue. If the order of the packet sequence is always the same, the last clients in the sequence experience a higher packet loss than the other clients. Thus, the packets transferred by msend() must be randomly shuffled. This insight does not only apply to the CacheCast support in a file server but to the functionality of CacheCast in general. Therefore, the msend() system call should perform the random shuffling of the packets before transmission.

8.2 Open problems and future work

Throughout this thesis we have studied various aspects of file transmission and the CacheCast mechanism in order to build an efficient file server with CacheCast support. However, due to the time limit of this project some areas have not been studied. In this last section we present some open problems and propose some directions for future research on the CacheCast file server.

In the experiments performed in this thesis, the arrival rate of the clients to the server is constant over a short time interval. While this arrival rate pattern gives a good indication of the behaviour of the system, a more realistic arrival rate distribution, such as a Poisson distribution, could be used to further investigate how the system behaves under various conditions.

The Rate control evaluation has revealed an issue when using transmission of original blocks. If a block scheduled for transmission has already been received by a client, this client is not added to the set of clients to which to transfer the block. If this client is the currently fastest one, the transmis-

sion rate can not be adjusted to this client. Basically, this issue is related to the fact that the server does not always transmits to all clients when transmission of original blocks is used. If transmission of original blocks could be modified such that the server would always transmit to all clients, this issue would no longer be present. This is a topic for future research. A possible solution could be to schedule multiple blocks for transmission. In this thesis the server only schedules the transmission of one block, selected by the Round Robbin algorithm, to the clients missing it. However, if the server also scheduled more blocks for transmission such that all clients were satisfied, the server would transmit to all clients, and the Rate controller would be able to adjust the transmission rate to any client.

The CacheCast file server relies heavily on the functionality of DCCP, specifically the congestion control mechanism and the fact that the messages are treated as individual entities. However, DCCP has not yet been widely adopted on the Internet. Another transport protocol which preserves message boundaries is UDP, which is well supported on the Internet. Still, UDP does not include congestion control. In order for UDP to be used, a separate congestion control mechanism would have to be built on top of UDP. Prabhakaran et al. have developed a protocol called *Tornado Transport Protocol* [40] which uses UDP and incorporates a congestion control mechanism. Future work on the CacheCast file server could include a study of whether UDP is a viable alternative for a transport protocol.

As discussed in Section 6.7, in order to get a full understanding of the performance of the CacheCast file server, the computational complexity of the system should be taken into account - especially the encoding and decoding when using Fountain codes. This could be achieved by building a real-world system and evaluating its performance on real hardware.

Bibliography

[1] DCCP in ns-3 through the Network Simulation Cradle. [ONLINE]. http://gitorious.org/ns3-nsc-dccp.

[2] Download statistics for Apache OpenOffice. [ONLINE]. http://sourceforge.net/projects/openofficeorg.mirror/files/stable/3.4.1/stats/timeline?dates=2012-08-01+to+2013-06-01.

[3] Download statistics for Notepad++ Plugin Manager. [ONLINE]. http://sourceforge.net/projects/npppluginmgr/files/v1.0.8/stats/timeline?dates=2011-11-01+to+2013-01-01.

[4] Network Simulation Cradle. [ONLINE]. http://www.wand.net.nz/~stj2/nsc/.

[5] The ns-2 network simulator. [ONLINE]. http://www.isi.edu/nsnam/ns/.

[6] The ns-3 network simulator. [ONLINE]. http://www.nsnam.org/.

[7] VLC media player. [ONLINE]. http://sourceforge.net/projects/vlc/.

[8] J William Atwood, Octavian Catrina, John Fenton, and W Timothy Strayer. Reliable multicasting in the Xpress transport protocol. In *Local Computer Networks, 1996., Proceedings 21st IEEE Conference on*, pages 202–211. IEEE, 1996.

[9] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298.

[10] Robert Braden, David Borman, and Craig Partridge. Computing the internet checksum. *ACM SIGCOMM Computer Communication Review*, 19(2):86–94, 1989.

[11] J.W. Byers, M. Luby, and M. Mitzenmacher. A digital fountain approach to asynchronous reliable multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1528 – 1540, oct 2002.

[12] Bin Chang, Liang Dai, Yi Cui, and Yuan Xue. On Feasibility of P2P on-demand streaming via empirical VoD user behavior analysis. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pages 7–11. IEEE, 2008.

[13] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[14] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proceedings of the second ACM international conference on Multimedia*, pages 15–23. ACM, 1994.

[15] S.E. Deering. Host extensions for IP multicasting. RFC 1112 (Standard), August 1989. Updated by RFC 2236.

[16] C. Diot, B.N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *Network, IEEE*, 14(1):78 –88, jan/feb 2000.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[18] Victor Firoiu and Marty Borden. A study of active queue management for congestion control. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1435–1444. IEEE, 2000.

[19] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341 (Proposed Standard), March 2006.

[20] S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), March 2006. Updated by RFCs 5348, 6323.

[21] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill, Inc., 2002.

[22] Jim Gemmell, Jim Gray, and Eve Schooler. Fcast Multicast File Distribution. *Network, IEEE*, 14(1):58–68, 2000.

[23] Jim Gemmell, Todd Montgomery, Tony Speakman, and Jon Crowcroft. The PGM reliable multicast protocol. *Network, IEEE*, 17(1):16–22, 2003.

[24] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448 (Proposed Standard), January 2003. Obsoleted by RFC 5348.

[25] Joao P Hespanha, Stephan Bohacek, Katia Obraczka, and Junsoo Lee. Hybrid modeling of TCP congestion control. In *Hybrid Systems: Computation and Control*, pages 291–304. Springer, 2001.

[26] M. Hosseini, D.T. Ahmed, S. Shirmohammadi, and N.D. Georganas. A Survey of Application-Layer Multicast Protocols. *Communications Surveys Tutorials, IEEE*, 9(3):58 –74, quarter 2007.

[27] Kien A Hua, Ying Cai, and Simon Sheu. Patching: a multicast technique for true video-on-demand services. In *Proceedings of the sixth ACM international conference on Multimedia*, pages 191–200. ACM, 1998.

[28] Cheng Huang, Jin Li, and Keith W. Ross. Can internet video-on-demand be profitable? *SIGCOMM Comput. Commun. Rev.*, 37(4):133–144, August 2007.

[29] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *Proceedings of the 11th international conference on World Wide Web*, pages 293–304. ACM, 2002.

[30] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001. Obsoleted by RFC 5321, updated by RFC 5336.

[31] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006. Updated by RFCs 5595, 5596, 6335.

[32] Dong Lin and Robert Morris. Dynamics of random early detection. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 127–137. ACM, 1997.

[33] David J. C. Mackay. Fountain codes. *IEE Communications*, 152:1062–1068, 2005.

[34] M. Mitzenmacher. Digital fountains: a survey and look forward. In *Information Theory Workshop, 2004. IEEE*, pages 271 – 276, oct. 2004.

[35] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[36] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[37] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[38] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

[39] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[40] Vijayan Prabhakaran, Joseph Stanley, and Paul Barford. High throughput data transfers using the tornado transport protocol, 2002.

[41] Rob Sherwood, Ryan Braud, and Bobby Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 941–951. IEEE, 2004.

[42] Hassan Shojania and Baochun Li. Tenor: making coding practical from servers to smartphones. In *Proceedings of the international conference on Multimedia*, MM '10, pages 45–54, New York, NY, USA, 2010. ACM.

[43] P. Srebrny. *CacheCast: a system for efficient single source multiple destination data transfer*. PhD thesis, University of Oslo, 2011.

[44] P. Srebrny, T. Plagemann, V. Goebel, and A. Mauthe. CacheCast: Eliminating Redundant Link Traffic for Single Source Multiple Destination Transfers. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 209 –220, june 2010.

[45] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335.

[46] Gabor Szabo and Bernardo A Huberman. Predicting the popularity of online content. *Communications of the ACM*, 53(8):80–88, 2010.

[47] M. Watson. Basic Forward Error Correction (FEC) Schemes. RFC 5445 (Proposed Standard), March 2009.

[48] Brian Whetten, Todd Montgomery, and Simon Kaplan. *A high performance totally ordered multicast protocol*. Springer, 1995.

[49] Hongliang Yu, Dongdong Zheng, Ben Y Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 333–344. ACM, 2006.